AD-A115 616 AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
THE DESIGN AND IMPLEMENTATION OF A DATA FLOW MULTIPROCESSOR.(U)
DEC 81 M W BRAY
UNCLASSIFIED AFIT/GCS/EE/81D-5 NL

1 OF 2
AD A
115616

UNITED STATES AIR FORCE

AIR UNIVERSITY

AIR FORCE INSTITUTE OF TECHNOLOGY

Wright-Patterson Air Force Base, Ohio

1

AFIT/GCS/EE/81D-5

THE DESIGN AND IMPLEMENTATION OF A DATA FLOW MULTIPROCESSOR

THESIS

AFIT/GCS/EE/81D-5 Michael W. Bray
Captain USAF

AFIT/GCS/EE/81D-5

THE DESIGN AND IMPLEMENTATION OF A DATA FLOW MULTIPROCESSOR

THESIS

Presented to the Faculty of the School of Engineering
of the Air Force Institute of Technology
Air University
in Partial Fulfillment of the
Requirements for the Degree of
Master of Science

by
Michael W. Bray, B.S.
Captain USAF
Graduate Electrical Engineering

December 1981

Preface

In this investigation, I attempted to design and implement a data flow multiprocessor. My main motivations for attempting this effort were to learn about microprocessors and interfaces between processors in a multiprocessing system. Since I have been primarily a software specialist, this effort presented real challenges. Unfortunately, my background proved to be a hindrance when hardware problems were encountered. However, the solution to these problems proved to be a great learning experience.

I would like to express my deep appreciation to Dr. Gary Lamont, who as my research advisor patiently answered many questions and guided me through to the solutions of many hardware problems. Also, I wish to thank Captain Charles Papp who taught me how to use the logic analyzer and the storage oscilloscope. Without these tools, I could never have debugged and repaired the microprocessors. Finally, I wish to thank my thesis readers, Major Charles Lillie and Major Walt Seward, for taking valuable time to wade through yet another thesis, and to provide comments which helped to improve its quality.

Michael W. Bray

Contents

List of Figures

Figure Page

List of Tables

AFIT/GCS/EE/81D-5

Abstract

This report presents a data flow multiprocessor designed and implemented with standard laboratory microcomputer boards. Intel SBC 80/20 small board computers were used since their design supported a multibus structure required for multiprocessing. Current data flow techniques were researched in order to find a technique that could be implemented through software. A packet communication architecture was chosen for implementation since other data flow techniques required specialized hardware.

The requirements of the multiprocessor were defined using structured analysis techniques. These requirements were then translated into structured modules. The software modules were then implemented and tested in a top down approach. The data flow multiprocessor software was tested by executing complete data flow programs. The results of the tests demonstrated the functionality of the multiprocessor. However, the multiprocessor software was limited in some respects. The memory allocated for the storage of data flow programs limited the maximum number of data flow instructions that could be represented to only 128. The mathematical operations were also limited in that only 8 bit coomputations were allowed.

I. Introduction

There has been increased interest in data flow computation in the past five years. Because of its great potential, data flow research is presently being conducted at over a half dozen universities and research labs in the US and Europe (Ref 12:49). The major emphasis lies in developing processors capable of processing data flow programs. The reason is that data flow computers offer a cost effective method of exploiting concurrency of computations on a large scale (Ref 12:48). In order to gain a better understanding of current data flow research, this investigation has attempted to apply current techniques to the implementation of a data flow multiprocessor system. The multiprocessor system was designed to use commercially available microprocessors with software providing the interface between processors.

Historical Perspective

In the past, large scale computers have been considered useful for many types of computing tasks. However, the costs of large systems in terms of hardware and software is becoming increasingly prohibitive when compared to the falling cost of microcomputers. Therefore, it has become more desirable to shift from large timeshared machines (such as the CDC 6600) to smaller processor systems (VAX 11/780, PDP 11/series, etc.). For some applications, it has even become practical to use microcomputers to perform computational tasks. However, due to their limited processing speed, microcomputers are limited to the types of tasks they can solve. Some highly compute-bound or time critical calculations often can not be performed by microcomputers within specified time constraints. To solve highly computational tasks using microcomputers, it would be desirable to have a practical way to increase the power of a microcomputer system incrementally by adding processors. Specialized processors have been employed in many large multiuser systems to increase computational power. However, multiprocessing of this type has rarely been used with microprocessor systems.

Background

The problem of multiprocessing in a small microprocessor system as opposed to the large systems is the manner in which the job function is divided among the processors. In large systems, there are generally several users simultaneously using the system. Each of the users' work is effectively independent of the others. Therefore, it is not difficult to schedule these independent tasks in a multiprocessor. Small microprocessor systems, on the other hand, are typically used in small dedicated applications where a computer serves one user. To effectively multiprocess, the computer must divide te users' work between processors. That is often difficult because the typical user considers his work to be comprised of a sequence of logical tasks done one at a time, not as a set of actions that may be done in parallel. Therefore, to effectively multiprocess on a single user microprocessor, the logical tasks requested by the user must be partitioned into a set of noninterfering subtasks.

There are several methods of partitioning the tasks into subtasks. One method of partitioning is to divide the work into functional units. This means that one processor operates the terminal, one the disk drive, and one is the computer which actually performs the intended task. Obviously, this approach works nicely only for tasks which are highly input/output bound. If the task is compute-bound, this approach wastes processors and returns little or no speed improvement. Another method of partitioning is to allow the user to specify independent program areas that can be executed simultaneously. This approach has merit in that it is relatively easily implemented on most computer systems and it will often provide significant parallelism if the programmer takes full advantage of the machine. The user, however, has the complex job of determining if a section of a user task is independent of another section. This job is difficult for even the most experienced programmers. Therefore, this method will not be considered because it is impractical to expect a general user to be able to partition his work into independent sections. The remaining method of partitioning is to allow the processor to determine those sections of the user's task that can be executed in parallel. One way of representing this is called data flow processing (Ref 8).

In data flow representation, each task is considered as a directed graph where the arrows represent movement of data and the nodes represent data transforms. Each transform is completely independent of all others because it depends solely on the arrival of data on its inbound arcs. This independence allows the individual transforms to be executed arbitrarily as soon as all of their input values are present. Also, transforms can be allocated arbitrarily to processors of a multiprocessor system. When done with its computation, the node places the resultant value on the outbound arcs for transfer to other nodes.

Figure 1 illustrates the computational flow through three data flow nodes. Nodes 1 or 2 can execute independent of each other whenever they have received their required operands (input values). When they complete execution of their operations (addition and multiplication respectively), they pass their result values to node 3, which is an addition node. When both inputs are present at node 3, it executes and passes its result to its only output arc. In this example, the value 9 would be passed to the output arc of node 3.

Today, there is much interest in developing machines that can directly execute a data flow representation of a problem. There have been several designs of large data flow processors proposed in the literature. For example,

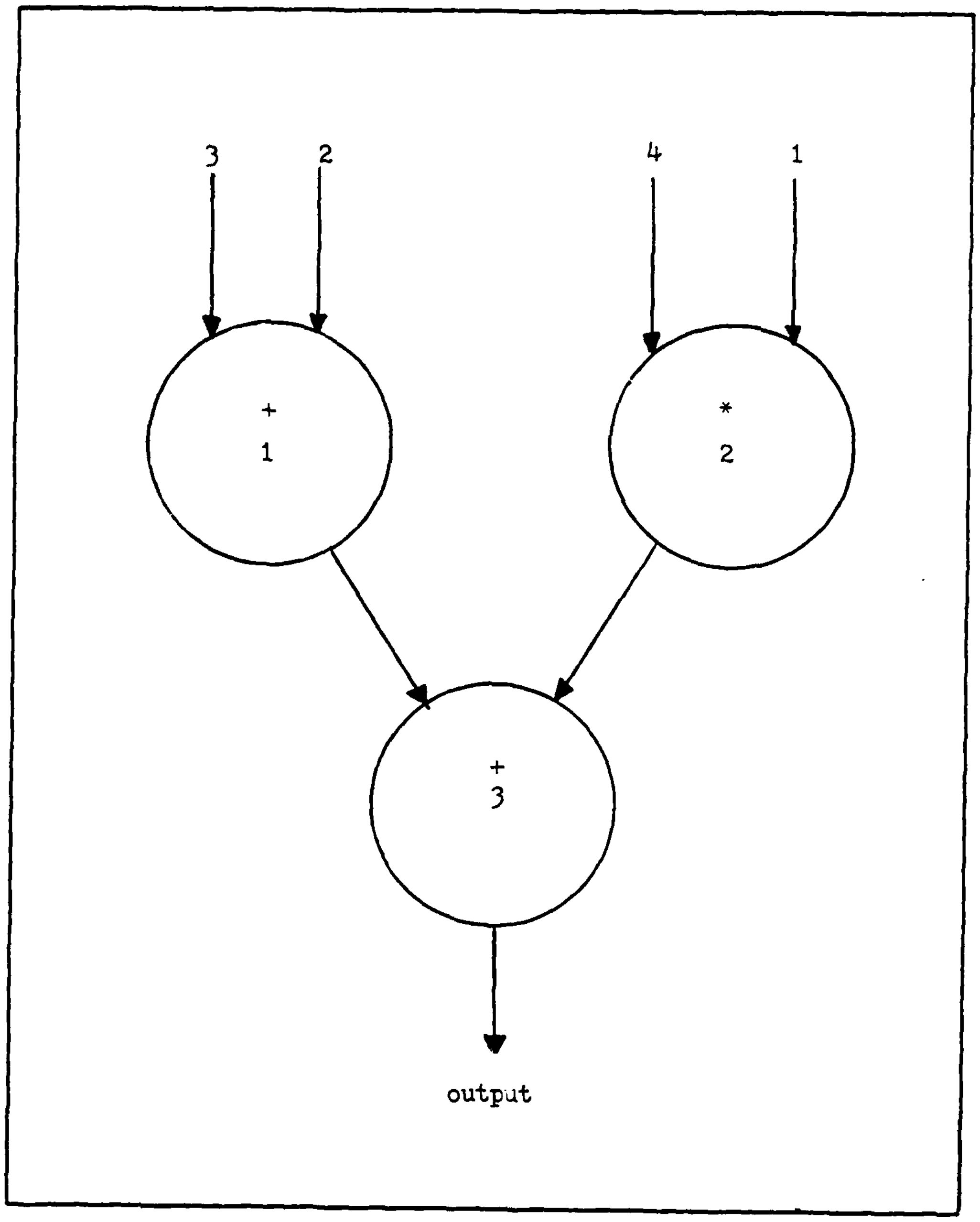


Fig. 1 Sample Data Flow Representation

Rumbaugh has proposed one multiprocessor architecture that will execute data flow (Ref 24). Arvind and Gostelow proposed a machine which consisted of hundreds of small asynchronously operating processors (Ref 3). Each of these processors accepts and performs a small task generated by a data flow program, produces partial results, and then sends these partial results to other processors in the system. Lawrence Livermore Laboratories has done simulation studies on the practicality of data flow computer systems that would theoretically be capable of many times the throughput of large conventional systems (Ref 1).

Though most of the data flow research seems directed at the large processor, it would appear that data flow techniques could be applied to the single user multi-microprocessor system. In such a system, the intent would be to use parallelism to increase the execution speed of the microprocessing system to eliminate the need for high cost large scale processors. This is, in fact, the motivation for this research effort. This effort follows a graduate research project addressing the implementation of data flow techniques in a single user multiprocessor system. Brian Boesch conducted this research and was partially successful in implementing a data flow multiprocessor (Ref 6). Chapter III presents in greater detail the efforts of Boesch, in addition to past and present data flow techniques.

Objective of This Investigation

The objective of this investigation was to research current data flow techniques and to apply these techniques to the design and implementation of a single user multiprocessor system. The specific goal was to understand current data flow techniques in sufficient detail such that a data flow multiprocessor could be implemented using laboratory microcomputer resources.

Approach

The design and implementation approaches taken in the literature started with the development of a data flow language such as ID-Irvine Data Flow (Ref 3), VAL (Ref 1), or Data Flow Procedure Language (Ref 11). The next step was to define the requirements for a processor design which would process the desired data flow language. Finally, the last step was to actually implement the data flow processor. A similiar approach will be taken in this investigation. However, the data flow processor will be designed using existing processors.

Prior to formulating a design and implementation approach, a thorough investigation of the literature was conducted. This was done to gain a working knowledge of current data flow languages, and architectures, as well as the benefits, limitations, and problems of data flow computing. Once a sufficient background had been gained, a design and implementation approach was formulated.

The first step was to specify the functional requirements for a data flow processor, in terms of both hardware and software requirements. These requirements are presented in Chapter IV. The software requirements were documented using Structured Analysis (Ref 26) to aid in the presentation of the software design. Since this effort is actually a software representation of a hardware design, the actual data flow processor was designed using structure charts and structured English (Ref 26). These techniques result in clear written specifications and diagrams that provide unambiguous design information.

The second step of the approach was to select a data flow language. This step is much more involved than merely selecting a language from the literature and implementing it. In the literature, a language was designed first, and then hardware was designed to process the language. One of the hardware requirements of this research effort was that the data flow processor be implemented using current laboratory microcomputers. Therefore, a data flow language had to be formulated rather than selected from languages in the literature. Chapter IV describes the requirements for the language, and Chapter V describes the language designed.

The last step was the actual implementation of the design. In this step, the functional requirements were converted into the data flow multiprocessor. The design was implemented in a top-down fashion (Ref 26) to aid in software testing and debugging.

Overview of the Thesis

The structure of the thesis basically follows the approach that was taken in the investigation. A discussion of data flow benefits, limitations, and problems are described in Chapter II. A brief explanation of data flow techniques pertinent to this effort is presented in Chapter III. This is presented to provide an understanding of current techniques, and to provide a background of previous research related to this investigation.

Chapter IV translates the functional requirements into hardware and software requirements. Structured Analysis is described and data flow diagrams are used throughout the chapter to support the written description of the software requirements.

The design of the data flow multiprocessor is presented in Chapter V. The design is described using structure charts and structured English. The code for the multiprocessor software is in Appendix A. Appendix B contains a user's guide for the software and discusses the important hardware idiosyncrasies.

Chapter VI describes the implementation and testing of the data flow processor. In this chapter, the factors affecting implementation are described as well as the testing techniques employed.

Finally, Chapter VII summarizes this investigation and gives recommendations for follow-on research efforts.

II. Data Flow Concepts an Overview

Introduction

This chapter presents a brief overview of the benefits of and problems with data flow computing. The first section compares the conventional von Neumann/Babbage architecture with the a data flow architecture. The next section describes the benefits of data flow computing. The following section describes some of the problems with data flow computing. Finally, the last section discusses some possible applications of data flow architectures.

Comparison With von Neumann Model

When attempting to adapt conventional von Neumann/Babbage machines for parallel computation, one discovers two particularly troublesome attributes of the von Neumann/Babbage model (Ref 3):

-centralized sequential control

-memory cells.

Sequential control is troublesome since it prohibits the asynchronous behavior and distributed control of a parallel processor. It also burdens the programmer with the need to explicitly specify (or to use an analyzer to determine) exactly where concurrency is to occur. The second point, the memory cell, presents a difficulty since its existence forces the programmer to consider not only what value is being computed, but also where that value is to be kept.

This places additional burden on the programmer and presents problems in program verification. Furthermore, in an asynchronous environment memory cells become even less tolerable. This is illustrated by an extreme case: the global variable. In this situation there are some otherwise asynchronous processor modules busy executing tasks, which must be coordinated through a common cell. This situation calls for complex synchronization controls to insure orderly use of the global variable. Such controls are difficult to design into a machine and may be very costly in execution time. Synchronization controls are also tedious for programmers to use, especially where large numbers of activities are to be coordinated.

The two troublesome attributes of the von Neumann model are not present when using data flow techniques. The data flow language is everywhere asynchronous except where synchronization is explicitly specified (i.e. no sequential control), and values are the subject of computation rather than the places where those values are kept (i.e. no memory cells). An asynchronous language, such as data flow, assumes computations are unrelated, and thus concurrent, unless otherwise specified. A sequential language on the other hand requires explicit specification to identify those places where concurrent processing may be initiated. The absence of memory cells insures that only simple control mechanisms are needed to coordinate access to data. Such a language should work well with a machine composed of multiple cooperating processors. The multiprocessor in this investigation is this type of machine.

Benefits of Data Flow Computing

Data flow architectures exploit parallelism on a global basis through the use of a machine level program representation based on the concept of data flow (Ref 13). In such a machine instructions are activated by the arrival of data. Instructions "fire" only when all of their input operands are available. The completed instruction then passes the result on to instruction(s) which require that result to fire. This is illustrated in Fig 2a by the arrival of the values 5 and 3 at the addition node. This node is enabled and fires passing its result to the multiplication node. Many of these instructions can be activated concurrently, or asynchronously. Since instructions are data-driven and can execute concurrently, a data flow representation of a problem can drastically speed up the execution time.

Another benefit of using data flow techniques is that the data flow language is purely functional and produces no side-effects as a result of its execution (Ref 1:2-3). A functional data flow program means that a unique set of output values will be determined by any set of input values (Ref 11:7).

To understand what is meant by side-effects, existing numerical computation languages, such as FORTRAN, must be examined. These languages reflect the storage structure of the von Neumann/Babbage concept of computer organization in that each language has some method of effecting a change in the state of memory (Ref 1:2). In contrast, data flow languages are entirely free of side-effects. Each data flow instruction corresponds to a mathematical function and the entire effect of putting two instructions together is to compose another mathematical function. Data flow instructions generate values rather than memory locations.

Problems With Data Flow Computing

Though data flow appears on the surface to be the salvation for the single user parallel processing system, there are problems. For example, data flow programs are difficult to understand and the language is hard to program. Another problem is getting the data flow representation of a program into the memory of a data flow processor. There is also the problem with input/output. Even though data flow may execute a task faster, it will still be constrained by the time it takes to process input/output. It seems wasteful to use many processors to gain an execution speedup and then sit and wait for the I/O to complete. Finally, a data flow program could possibly be indeterminant, meaning that a given set of inputs could produce different outputs.

Difficulty of Understanding. The level of complexity of data flow programs are higher than programs of structured languages such as ALGOL or PASCAL. This is due in large part to the inherent concurrent execution of data flow programs. Programmers have grown accustomed to the top-down sequential flow of most programming languages. Therefore, the solution would be to program in a structured language such as PASCAL, and translate the source into a data flow machine language. It has been proven that all ALGOL-like programs can be translated to data flow (Ref 27). The translator, in addition to solving the difficulty of understanding the problem, will have the following effect: decreased errors, increased portability, and increased programmer acceptance.

Data flow is not inherently more error prone than other programming languages. However, humans cannot easily understand the maze of arcs and operators present in even a simple flow graph. This inability promotes errors. By hiding the data flow nature of the processor from the programmer, the use of a high order language (HOL) allows the programmer to use a language with which he is familiar.

There is a wealth of existing software today, most of it written in some HOL. If a compiler existed that translated from an HOL to data flow notation, some existing software could be used in data flow processors without costly rewritting.

Most programmers of today are familiar with one or more of the standard HOL's, such as PASCAL, ALGOL, or FORTRAN. Even though data flow techniques may be more efficient for

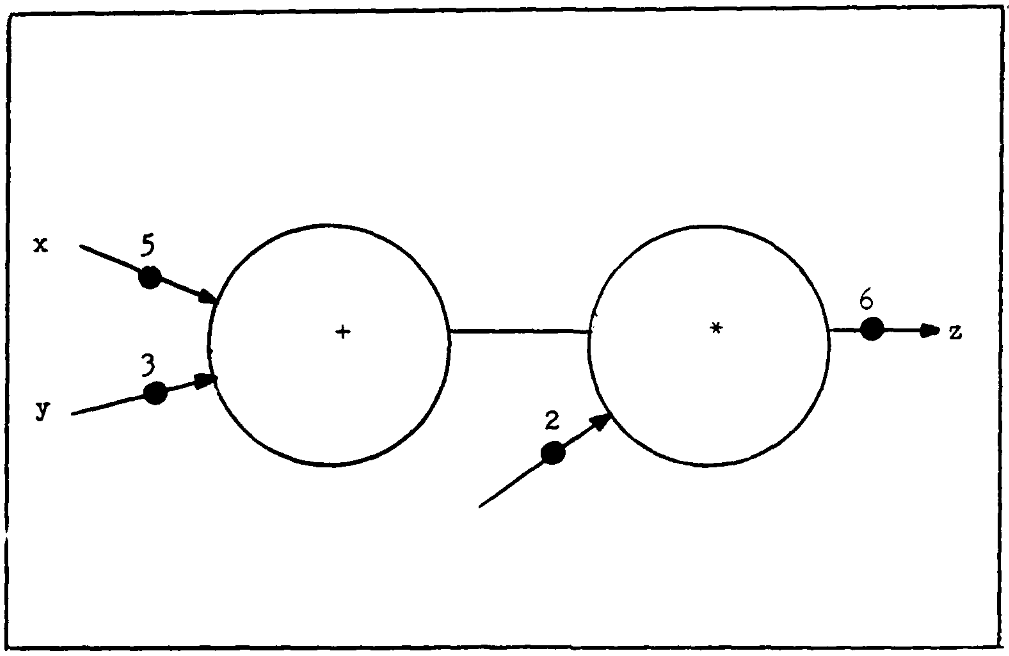


Fig. 2a Data Flow Indeterminancy

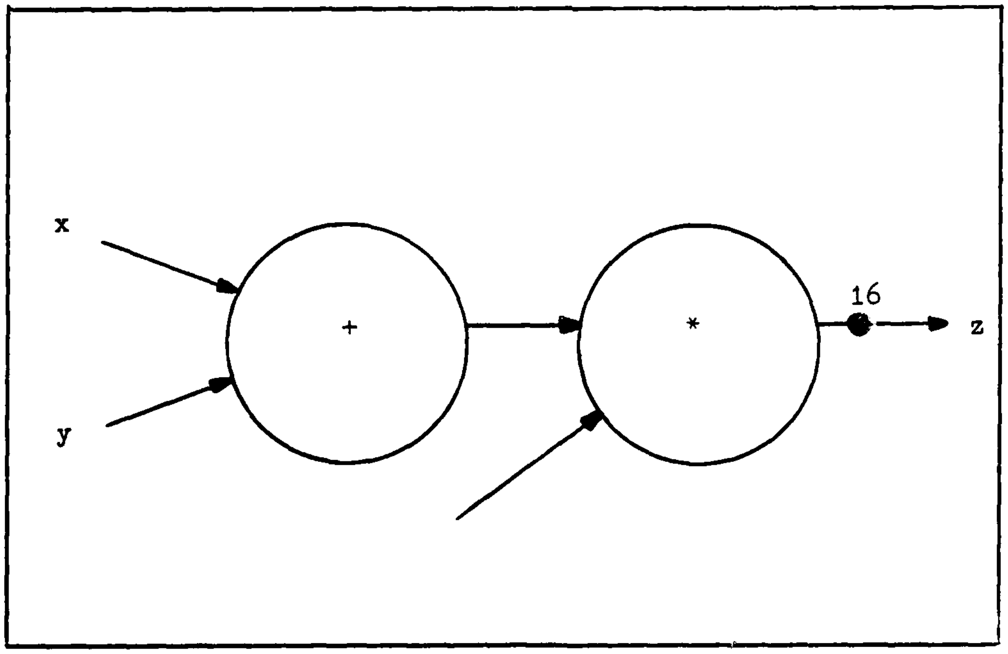


Fig. 2b Firing of Data Flow Nodes

certain applications, there is still a strong tendency to program in familiar languages. But by hiding the flow nature of the machine, the programmer will be able to continue programming in a language he is familiar with while having the advantages of data flow.

Input/Output. Assuming that data flow processors could operate in the nanosecond range, they would have to sit idle and wait for even the fastest I/O devices which operate in the microsecond range. For highly I/O bound programs, this waiting could waste processor time. One solution would be to use double or triple buffering. In other words, fill one buffer with incoming data while the processor is processing one or two buffers of data. However, even this solution would still waste some time sense the buffers would be processed prior to the I/O device filling the other buffer.

Indeterminancy. In a data flow program, when a set of input values does not produce a unique set of output values, the program is indeterminant. This is best illustrated by an example. Figure 2a shows a simple data flow program which adds two values and multiplies the sum by two. the large black dots on the input arcs as well as the "z" output arc represent tokens, which are similiar to Petri Net tokens (Ref 22). The tokens mean that an input or output value is present on an arc. In this case, the value 6 has been computed but not yet used. Some kind of node using the "z" input arc must fire before the 6 token can be removed. However, both x and y arcs have tokens present. This means that the addition node can fire. If the addition node were enabled and fired before the 6 token was used, the value of 16 could be placed on the z arc and destroying the previous value of the calculation (6). This is illustrated in Figure 2b. This problem can be solved through the use of acknowledge signals which control the firing of the nodes. This will be explained in greater detail in Chapters III and IV.

Applications

Data flow computing techniques lend themselves extremely well to almost any type of compute-bound task. This is because data flow programming languages have cleaner mathematical semantics than ordinary computational languages. Also, because data flow is basically applicative in nature and local in effect, the functions act solely on the data without states, continuations, or other complications (Ref 20:40). Tasks which are particularly amenable to data flow computing are the solution of differential equations, Fast Fourier Transforms, and many complex repetitive physics problems, such as those processed by Lawrence Livermore Laboratories (Ref 20). These types of tasks are highly repetitive and require great amounts of execution time. Since much of the computation in these tasks is concurrent, they could be processed using data flow computation and thus speed up the execution time. In simulation studies, Lawrence Livermore Laboratories was able to achieve a five-fold speedup in execution on one of their complex physics problems (Ref 20:17).

Data flow programs are also well suited to represent signal processing computations such as waveform generation, and filtering in which a group of operations is to be performed once for each sample of the signals being processed. Data flow techniques could be used to perform real-time laboratory monitoring, and to solve aircraft collision avoidance and related avionics problems. In short, data flow techniques could be used to solve any compute bound, repetitive scientific problem.

Summary

When adapting von Neumann/Babbage machines for parallel computation, one discovers two troublesome attributes: centralized sequential control, and memory cells. These attributes are not present when using data flow techniques since the data flow language is everywhere asynchronous, and values are the subject of computation rather than memory locations. The use of data flow techniques has the benefits of exploiting parallelism, thus speeding up the execution time, and the benefit of programming in a purely functional and side-effect free language. The use of data flow computing is not without its problems. For example, data flow programs are difficult to understand, the processing speed is constrained by the speed of the I/O device for I/O bound programs, and data flow programs may be indeterminant.

Applications which are highly compute bound and have some parallel computation, lend themselves to data flow computing. These types of tasks can be executed much faster, perhaps even up to a five-fold increase by using data flow techniques.

III. Current Data Flow Techniques

Introduction

The idea of data-driven instruction execution has appeared several times in the literature. Several hypothetical machines that implement specific data flow languages have been described (Ref 4,5,7,10,12,13,14,19,21). This chapter will discuss some of these architectures, since the data flow multiprocessor developed borrows heavily from their architectural techniques. Also, the chapter will provide a brief history of the previous data flow research effort conducted by Boesch (Ref 6). The intent of this chapter is not to restate the work of other researchers, but rather to furnish a background for an understanding of the architecture chosen to be implemented.

The first section of the chapter discusses some of the architectures that appear in the literature. Only the architectures which are pertinent to this investigation will be discussed. The architectures are presented in chronological sequence from the earliest to the latest. The next section presents a more detailed discussion of the work of the previous graduate research effort; that conducted by Boesch.

Architectures

This section presents two architectures proposed by Dennis (Ref 10,12,13,14): the Elementary Data Flow

Processor, and the Circular Pipeline Architecture.

Elementary Data Flow Processor. The Elementary Data Flow Processor was an early version of a data flow architecture proposed by Dennis of MIT (Ref 13,14). This machine avoided the problems of processor switching and processor/memory interconnections present in attempts to adapt conventional von Neumann/Babbage-type machines for parallel computation. Sections of the machine communicate by transmission of fixed-size information packets. The machine is organized so that the sections can tolerate delays in the packet transmission and still effectively utilize the hardware. The elementary processor presented here is designed to utilize a data flow language as its base language.

Figure 3 shows a block diagram of the elementary data flow processor. A data flow program to be executed is stored in the _Memory_ of the processor. The memory is organized into _Instruction Cells_, each cell corresponding to an operator of the data flow program. Figure 4 shows the format of an instruction cell. The cells are composed of three registers. The first holds an instruction which specifies the operation to be performed and the address(es) of the register(s) to which the result of the operation is to be directed. The second and third registers hold the operands for use in execution of the instruction. When a cell contains an instruction and the necessary operands, it is enabled and signals the _Arbitration Network_ that it is

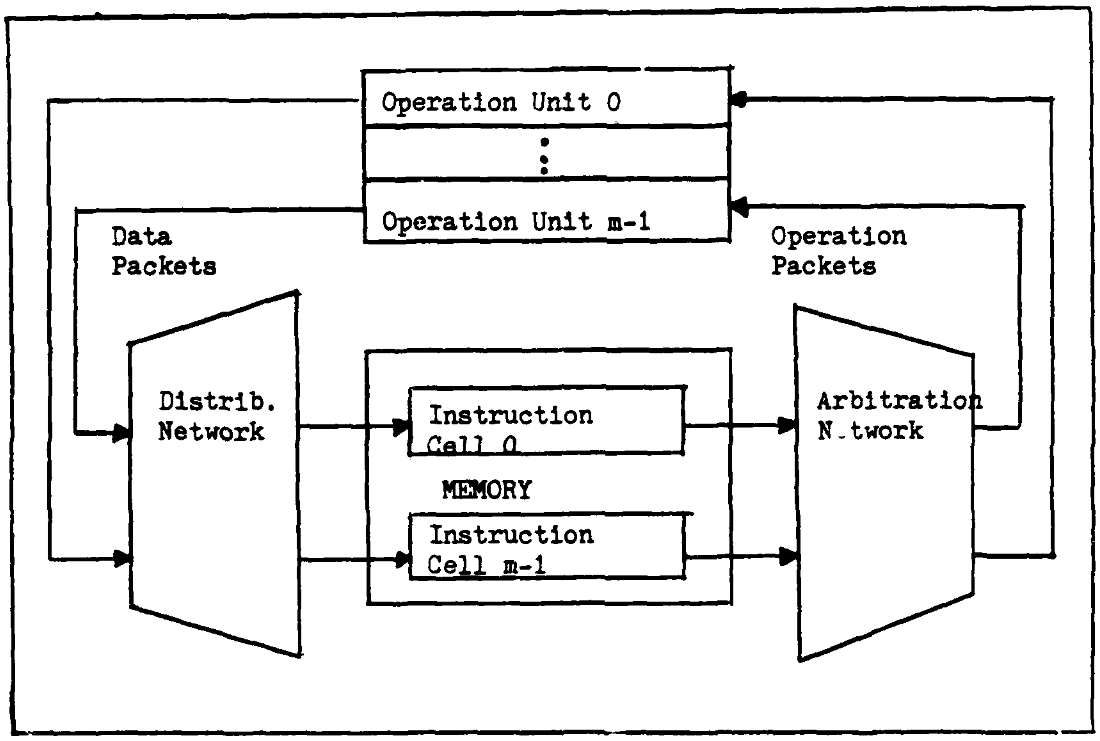


Fig. 3 Elementary Data Flow Processor

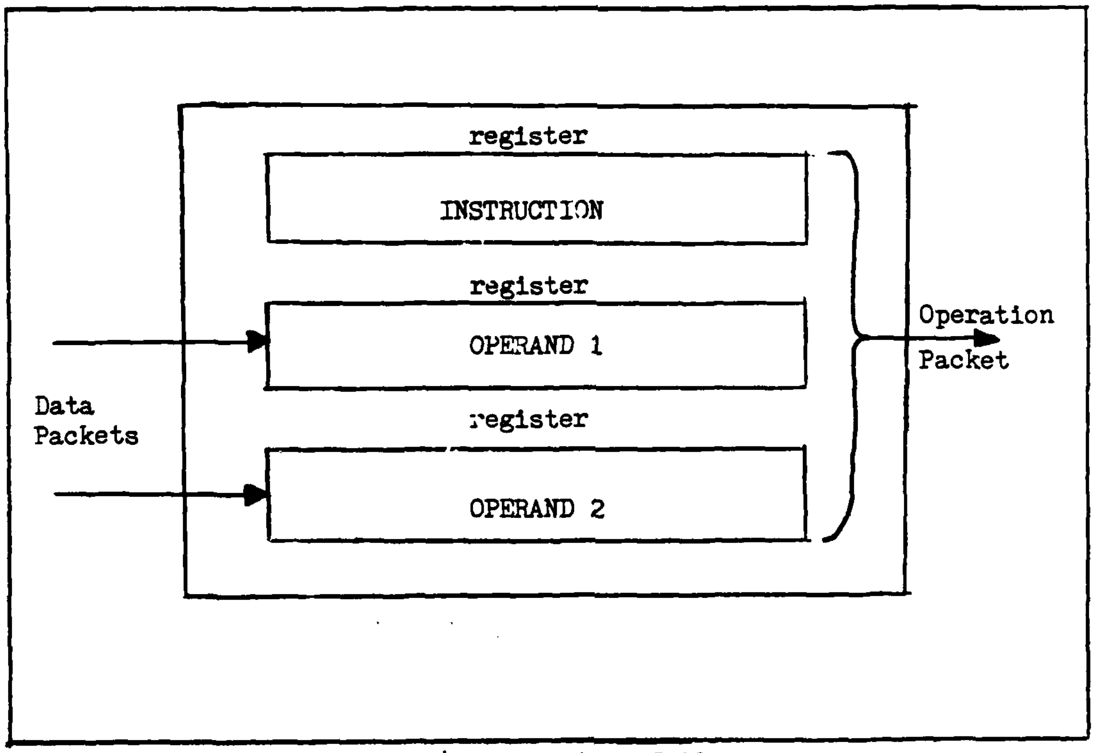


Fig. 4 Instruction Cell

ready to transmit its contents as an Operation Packet to an Operation Unit which actually performs the desired function. The arbitration network directs the operation packet to an appropriate operation unit, such as the addition unit, subtraction unit, multiplication unit, etc. The result of an operation leaves an operation unit as one or more Data Packets, consisting of the computed value and the address of a register in the memory to which the value is to be delivered. The Distribution Network accepts data packets from the operation units and directs the data item through the network to the correct registers in memory. The instruction cell containing that register may then be enabled if an instruction and all operands are present in the cell. The elementary processor just described is an example of a packet communication architecture.

Circular Pipeline Architecture. This architecture was designed to solve the problem of indeterminacy, as will be explained later. In this architecture (also a packet communication architecture), a data flow program is a collection of activity templates, each corresponding to an actor of a data flow program graph (Ref 9,12). An activity template corresponding to the plus operator is shown in Figure 5. There are four fields: an operation code specifying the operation to be performed; two receivers, which are places waiting to be filled in with operand values; and destination fields (in this case one), which specify what is to be done with the result of the

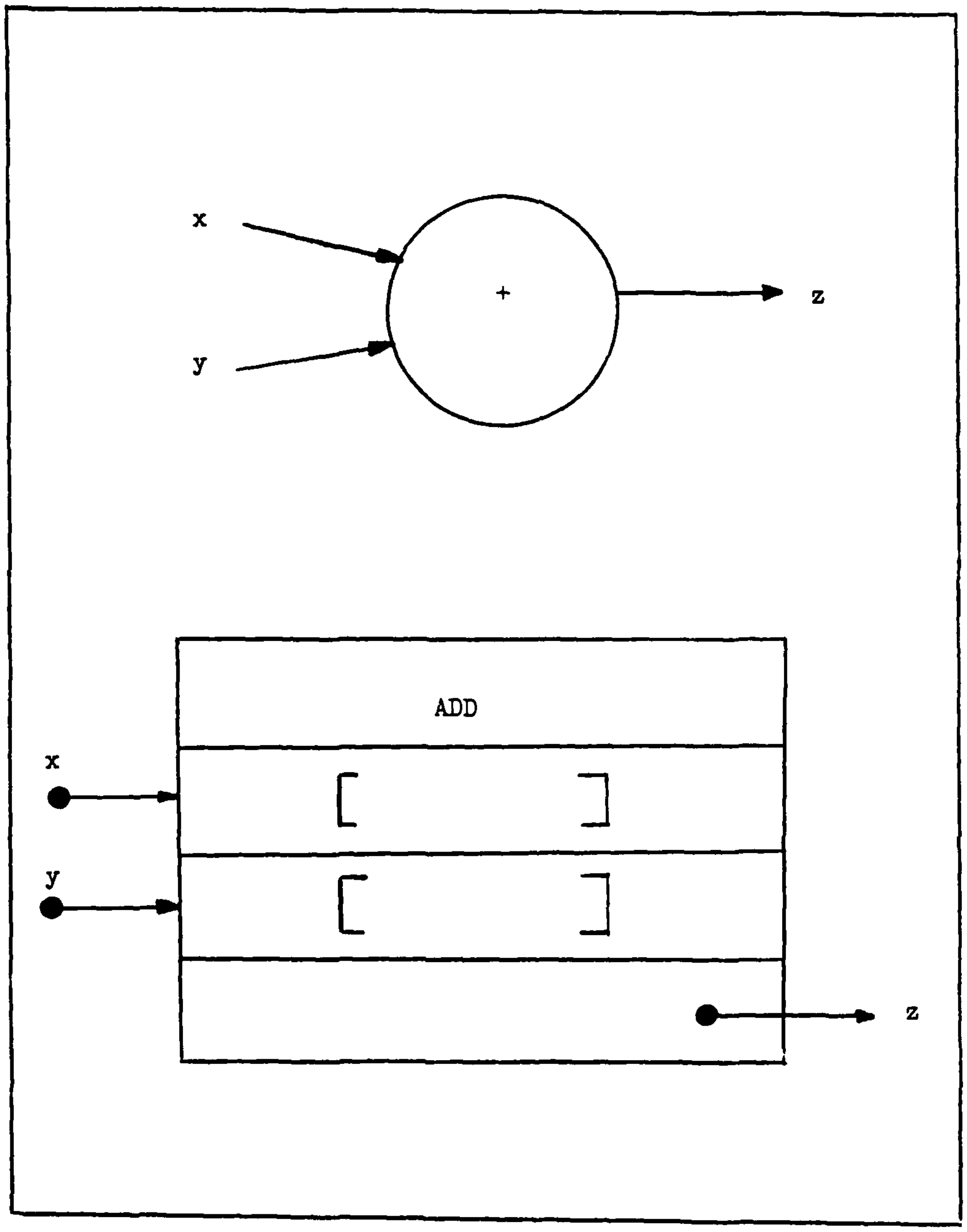


Fig. 5 Plus Operator Activity Template

operation on the operands.

An instruction of a data flow program is the fixed portion of an activity template. It consists of the operation code and the destinations; that is

instruction = <opcode, destinations>.

Figure 6b shows how activity templates are joined to represent a program, specifically the composition of operators in Figure 6a. Each destination field specifies a target receiver by giving the address of some activity template and an input integer specifying which receiver of the template is the target; that is

destination = <address, input>.

Execution of a machine program consisting of activity templates is viewed as follows. The contents of a template activated by the presence of an operand value in each receiver take the form:

operation packet = <opcode, operands, destinations>.

Such a packet specifies one result packet of the form

result packet = <value, destinations>

for each destination field of the template. Generation of a result packet, in turn, causes the value to be placed in the receiver designated by its destination field.

To prevent indeterminancy (as described in Chapter II), instructions cannot be activated until their previous values have been used by subsequent instructions. For example, in Figure 2a, the add instruction must not be reactivated until its previous result has been used by the multiply

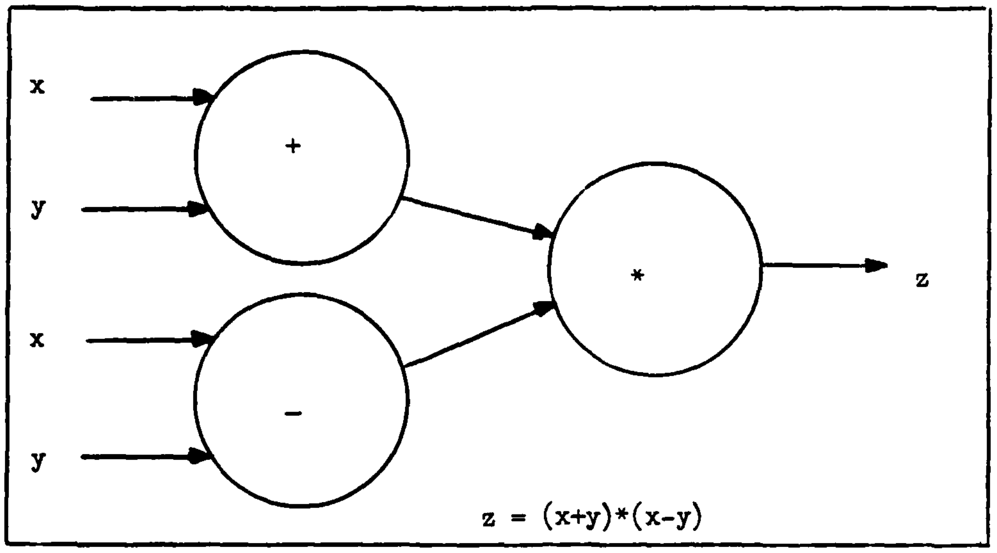


Fig. 6a Data Flow Computation (Operators)

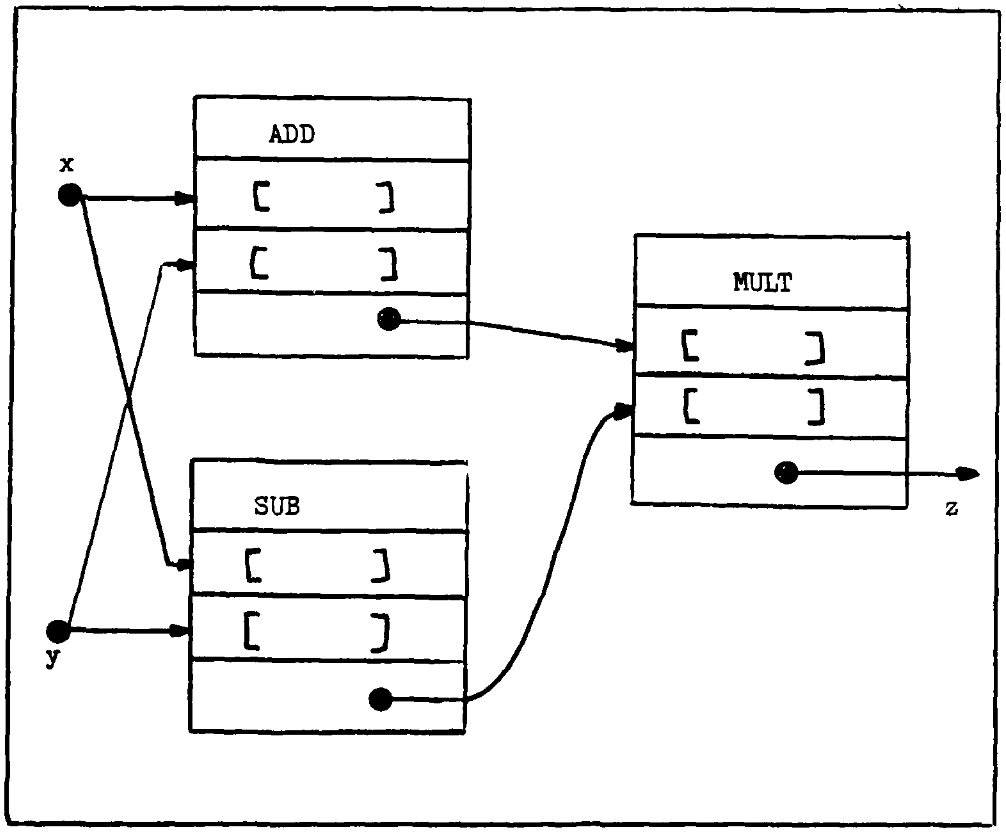


Fig. 6b Data Flow Computation (Activity Templates)

instruction. This constraint is enforced in this architecture through the use of acknowledge signals generated by specially marked designations (*) in an activity template. Acknowledge signals, in general, are sent to the templates that supply operand values to the activity template in question. Figure 7 illustrates the implementation of the data flow program of Figure 2a in terms of activity templates and acknowledge signals. With acknowledge signals, the enabling rule now requires that all receivers contain values, and the required number of acknowledge signals have been received. This number is written to the opcode field of an activity template.

The instruction execution mechanism used in this architecture is illustrated in Figure 8. The data flow program describing the computation to be performed is held as a collection of activity templates in the Activity Store. Each activity template has a unique address which is entered in the Instruction Queue unit when the instruction is ready for execution. The Fetch unit takes an instruction address from the instruction queue and reads the activity template from the activity store, forms it into an operation packet, and passes it on to the Operation Unit. The operation unit performs the operation specified by the operation code on the operand values, and generates one result packet for each destination field of the operation packet. The Update unit receives result packets and enters the values they carry into operand fields of activity templates as specified by

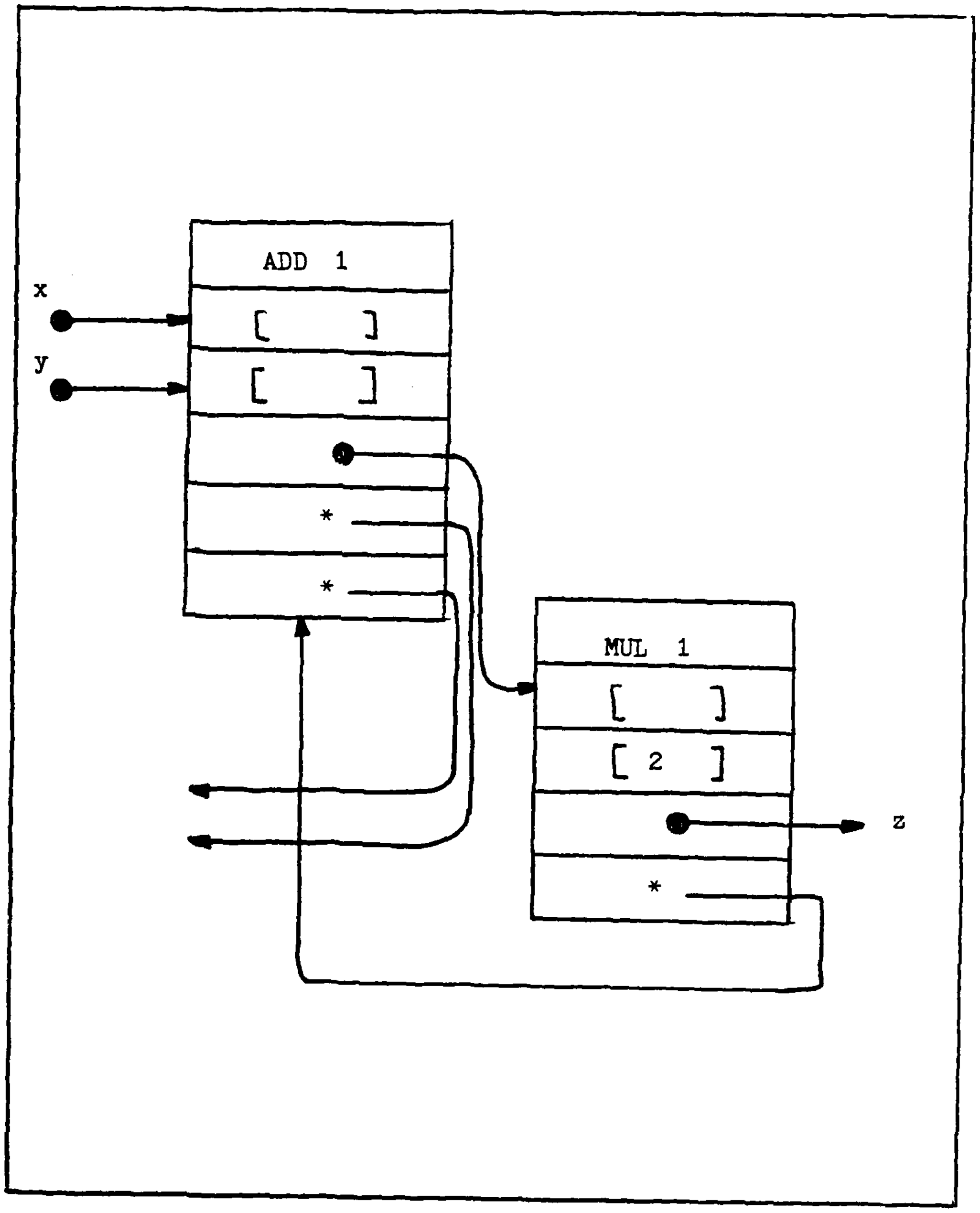


Fig. 7 Activity Templates Using Acknowledge Signals

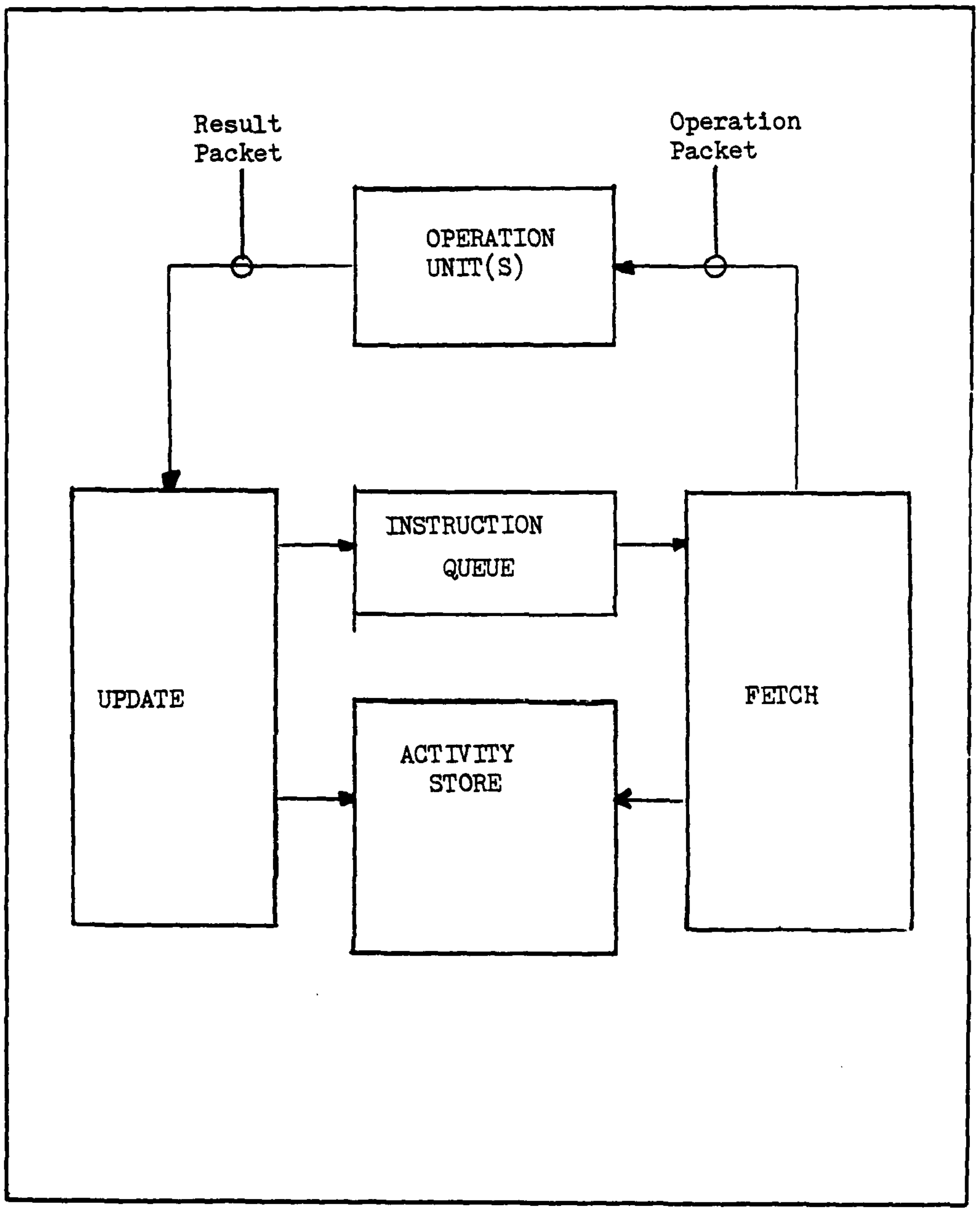


Fig. 8 Packet Communication Architecture

their destination fields. The update unit also tests whether all operand and acknowledge values required to activate the destination instruction have been received and, if so, enters the instruction address in the instruction queue. Execution continues in this manner until no instructions are in the instruction queue.

Boesch Investigation

As stated previously, this investigation is a continuation of a graduate research effort initiated by Brian Boesch (Ref 6). This investigation draws heavily on the techniques described in the Boesch thesis. In that investigation, simulation studies were conducted to verify that data flow techniques could be applied to the single user laboratory microcomputer system. The results showed that an execution speedup could be attained by using data flow techniques on certain concurrent computations. The execution speedup possible was not as dramatic as the fivefold speedup reported in the MIT and LLL simulations (Ref 20). This could be attributed to the fact that the computers simulated in the MIT and LLL studies were hypothetical data flow computers, as opposed to a simulation based on a conventional (von Neumann/Babbage model) microcomputer used in Boesch's thesis.

Since the simulation showed that an execution speedup could be obtained, the next step was to implement (through software) a data flow multiprocessor. The hardware used was two Intel SBC 80/20 microcomputers communicating via shared memory. The multiprocessor was designed to process a special data flow graph language derived in the thesis. The actual implementation was designed so that each of the two processors executed any enabled instruction (stored in shared memory), and placed the result in the proper instruction. The software that executed the multiprocessor was written in PASCAL and Z80 assembly language. Unfortunately, the hardware implementation did not work as planned, since the multiprocessor software was not fully debugged.

Even though the hardware implementation of the data flow processor did not work, much can be learned from this research. The most important being the manner in which the two processors communicated via shared memory. The communication method described in the Boesch thesis will be employed as explained in Chapter IV.

Summary

Several proposed data flow architectures appear in the literature. Of these, the architectures proposed by Dennis of MIT appear to contain techniques that could be applied. In particular, a packet communication architecture which processes a data flow program composed of activity templates could be emulated on a laboratory microcomputer system. The previous research effort has demonstrated, through simulation, that data flow techniques applied to microcomputers results in an execution speedup. The remainder of this report defines and implements an actual data flow multiprocessor.

IV. Functional Requirements

Introduction

With the exception of the past and present thesis efforts, all data flow research has been aimed at the design of a hardware implementation of a data flow processor. This investigation will be directed toward implementation of a data flow multiprocessor solely through the use of software. This chapter will define the functional requirements for the data flow multiprocessor.

The first section defines the operating environment of the multiprocessor system. The following sections present the requirements in terms of hardware and software respectively. Finally, the processor software requirements are specified using structured analysis techniques (Ref 26).

Operating Environment

Before specifying systems requirements, the operating environment must first be defined. For this investigation, the operating environment shall be defined in terms of the following: (1) single user, (2) laboratory environment, (3) reconfigurable.

Single User. The intent of this investigation is to determine ways to separate seemingly indivisible tasks into independent subtasks. Therefore, only one major task will be allowed to operate in the system at any given time. To allow multiprogramming on this system would complicate the design to such a degree that implementation would be near impossible!

Laboratory Environment. This requirement is primarily to define the type of tasks to be performed. The function of this system will be scientific numerical computations, not database processing or text editing, since these types of processing require text and character manipulation.

Reconfigurable. This means that the number of processors available at any given time should be hidden from the user. To the user, a three processor system should simply be faster than a two processor system. The intent here is to require program independence from any specific hardware parameters.

Pedagogical Requirement

One of the main motivations behind this thesis effort was the need to learn more about data flow techniques. Today, data flow research is being conducted at more than a dozen institutions (Ref 12). Before major data flow research can be conducted at AFIT, much has to be learned. Hopefully the efforts of the past and present AFIT theses will provide a foundation for future data flow research.

Hardware Requirements

Even though this effort is mainly concerned with the development of software, certain hardware requirements must first be detined. For example, most designs, including data flow architectures, place paramount importance on the requirement for fast execution speeds. The goal of this investigation is to successfully implement a data flow processor. Therefore, fast execution is a desired goal, but not necessarily a required goal. Future efforts will be concerned with faster execution speeds. Requirements that must be defined include: type of hardware, and communication between processors.

Type of Hardware. The hardware used in this investigation must be of the type found in the AFIT Digital Engineering Laboratory. The reason for this is twofold. First, due to the time limit placed on this investigation, valuable time can not be wasted waiting for hardware to be ordered and delivered. Second, the simulation studies conducted in the previous thesis were executed on an Intel microcomputer. To attempt to build a microprocessor board at this time is far beyond the scope of this effort, and should not be attempted till later.

The hardware should include: a CRT for communication with the system and sofware entry, a disk system for storage of the data flow processor sofware, two or more processors, and at least 64k bytes of RAM memory. The 64k bytes of RAM will be required since the RAM will be needed for shared memory, storage of the data flow processor code, storage of the actual data flow program, and storage of the host computer operating system software.

Communication Between Processors. Since this will be a multiprocessor system, there must be a way for the processors to communicate. Hayes classifies multiprocessors by the degree of coupling (communication) that exists between various processors (Ref 16). He lists the following three cases:

- Stand-alone computers,
- Indirectly Coupled, and
- Directly Coupled.

The directly coupled systems have the most communication between processors. For this design effort, a directly coupled multiprocessor using a master/slave processor relationship will be used. The reason for this choice is that a directly coupled system allows the greatest inter-processor communication, and a master/slave approach will allow the processors to act independently of each other.

Since there will be inter-processor communication, the microprocessors must be equipped with a multibus structure. This will permit the processors both to share RAM memory (for inter-processor communication), and to execute asynchronously. Both of these factors are needed in order to even approximate the design of a true data flow computer.

Software Requirements

The implementation of a data flow multiprocessor will require two kinds of software. One will be the software used to implement the multiprocessor. The other will be the actual data flow graph language. For the remainder of this report, the two types of software will be referred to as the processor software and the data flow language. Both types of software must be clearly defined prior to implementation since they are both mutually dependent on each other. The following sections present the requirements for the data flow language and the processor software.

Data Flow Language. This language will representthe actual data flow instructions. It must be understandable and efficient. Understandability is important because the human user must be able to look at the representation of a data flow instruction and determine the function of that instruction. Specifically, the user must be able to see the type of operation (opcode), the operands, and the destination addresses. Efficiency of the language is important because the entire data flow program must be stored in the limited RAM memory. For this investigation, the data flow multiprocessor will be implemented using 8-bit microprocessors. Because of this, only 64K can be addressed. However, the operating system and processor software require large portions of this 64K RAM. Therefore, only a small portion of RAM can be allocated to the storage of the data flow program. The next chapter specifies exactly how this memory can be allocated.

To accomodate the efficiency requirement, each instruction must be stored in contiguous memory locations, lying on 2,4, or 8 word boundaries. Contiguous storage will save memory in the sense that pointers and linked lists will not be required to point to various components of an instruction. Also, the data flow processor software will be less complicated if each data flow instruction requires the same amount of storage. Therefore, all instruction components must be stored in the same amount of memory. The components of the instruction must include: the node number (instruction number), the instruction type (opcode), the operand values, and the addresses where the results are passed. The next chapter will specify in detail the design of the data flow instruction and language.

The language must also prevent indeterminancy in the manner defined in Chapter III. Therefore, the language must have the capability to represent the use of acknowledge signals. The most efficient way to represent the use of acknowledge signals would be to treat each data flow instruction as an activity template as defined in Chapter III.

The language must, as a minimum, be able to represent integer calculations. Since this is an early attempt at the implementation of a data flow processor, real calculations will not be a requirement. Real number computations are affected by round off. These computations would be hard to program and to verify correctness. Integer computations, on the other hand, have no round off and are easier to program. Therefore, the language will initially represent only integer computations.

Processor Software. Since the processor software is what communicates with the processors, it must be more constrained than the data flow language. The processor software must be easily modifiable, understandable and well documented, efficient, and prevent race and deadlock conditions.

The software must be easily modifiable because it will be implemented in a top-down approach (Ref 26). Even though the use of the top-down technique minimizes errors, it would be unrealistic not to expect errors in the software. Procedures and functions must be able to be modified with little or no impact on previously tested and installed code. To meet the goal of modifiability, the system must be implemented in an efficient programming environment. Assembly language is not practical for all modules of the data flow processor software because the programmer has the burden of direct machine interface. However, some low level functions require the use of assembly language. Therefore, the software cannot be entirely programmed in a high order language (HOL). For this reason, a combination of programming languages will be required. Assembly language will be used to program some low level functions, and an HOL will be used to program the high level functions. PASCAL (Ref 28) will be the HOL since it supports structured programming. Specifically, UCSD-PASCAL (Ref 25) will be used since it contains an excellent programming development facility including program libraries, separate module compilation, and an interactive screen oriented editor for easy program modification. UCSD PASCAL also allows linkage to assembly language routines. FORTRAN and BASIC were rejected because neither of them offer the sophistication of the UCSD PASCAL programming environment.

The processor software must be understandable and well documented. Both of these constraints are needed for the benefit of future related thesis efforts. The generated code must be documented well enough so that future researchers can easily understand the code. As a minimum, each procedure must have in-line documentation describing the function of each variable and major computations.

The software must be efficient. As with the hardware, efficiency does not necessarily mean speed. Efficiency means that the object code must take up as little storage space as possible. The software will require a large amount of storage space and the RAM is limited in size.

Finally, the processor software must prevent race and deadlock conditions (Ref 18). These undesirable conditions could happen as a result of one processor writing to a shared memory location (thus blocking the other processor) while the other processor requests to read the same memory location. To prevent this, a lock byte (semaphore) (Ref 18) must be placed on shared memory when any of the processors read from or write to shared memory.

Structured Analysis Techniques

Structured Analysis is a technique using data flow diagrams (DFD's) and a data dictionary which uses Structured English, decision tables, and decision trees to describe the DFD's (Ref 26). The DFD's have the four components shown in Figure 9. The first component is the data flow. It is a pipeline of other data flows and of data elements. The data elements are the basic data types that cannot be partitioned further and still retain their meaning. The data flow is represented by a curved arrow on the DFD's. The process converts input data flows to output data flows and is the second component of the DFD's. It is represented by a circle or bubble that contains the process name. The boxes represent the third component of DFD's, the sources and sinks of information. They may represent a user keyboard, a CRT display, or any other mechanism by which information enters or leaves the system. Finally, files are the last component and are repositories of information within the system. They are shown by straight lines.

With this very brief discussion of DFD's, it is now possible to represent the functional requirements of the data flow processor software. Table I describes the process hierarchy for the multiprocessor DFD's. Figure 10 shows the high level DFD representation of the function of the multiprocessor software. The data flow multiprocessor is enabled via the CRT keyboard. Process 1 reads a data flow program from some disk file, then writes the program to

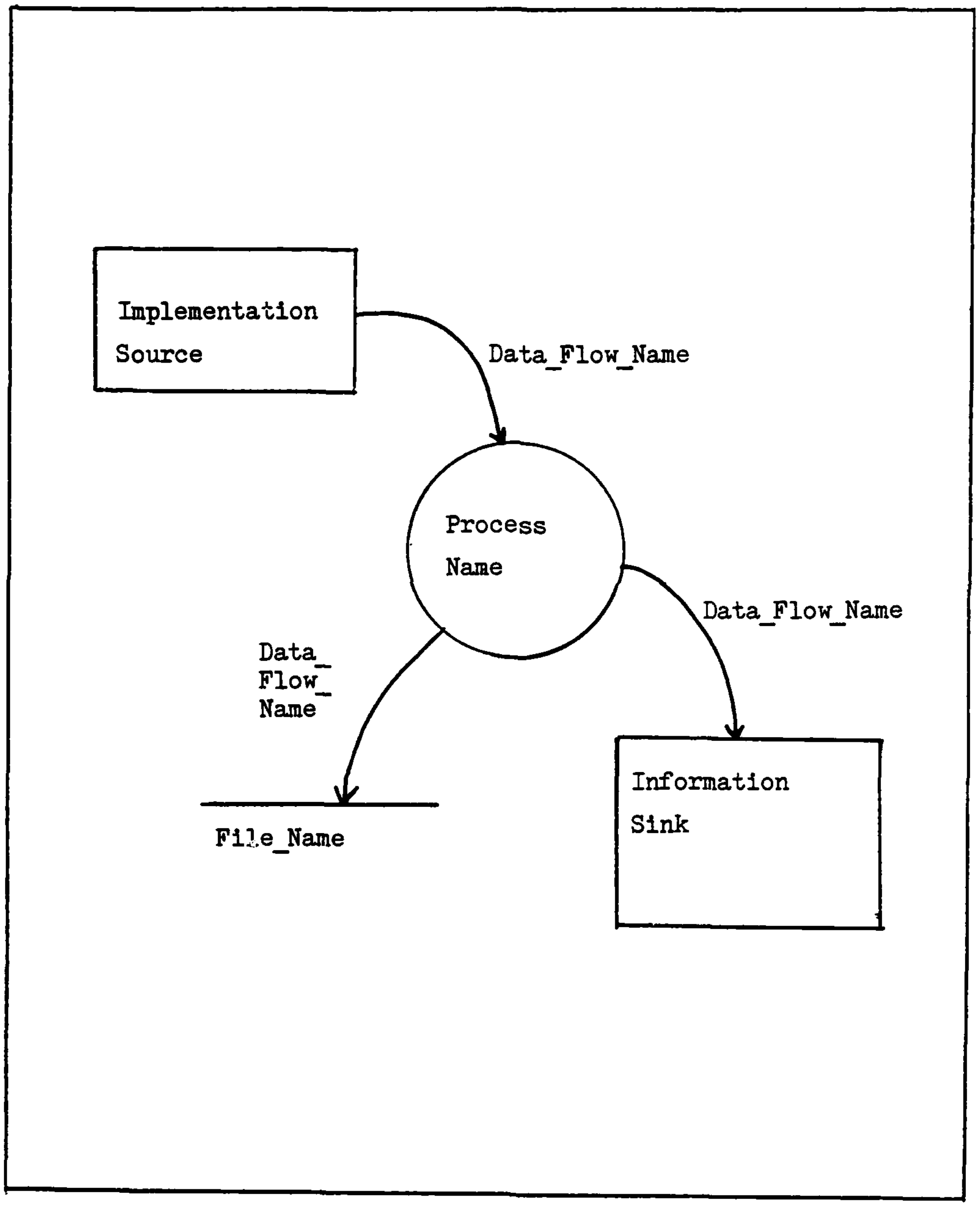


Fig. 9 Components of a Data Flow Diagram

Table I

<u>Multiprocessor Software Process Hierarchy</u>

1. Input Data Flow
   - 1.1 Write Blanks to Shared Memory
   - 1.2 Read Data Flow Program
   - 1.3 Fill Memory
     - 1.3.1 Break Down Components
     - 1.3.2 Convert to 2-Byte Values
     - 1.3.3 Compute Addresses
     - 1.3.4 Store Fields
   - 1.4 Enable Processors
     - 1.4.1 Read Processors Software
     - 1.4.2 Load Software
     - 1.4.3 Execute Software
2. Master Software
   - 2.1 Check for Enabled Nodes
     - 2.1.1 Read Instructions
     - 2.1.2 Check if #Ack. Sig. Req.=#Ack. Sig. Rec.
     - 2.1.3 Test if 1st Field is Blank
     - 2.1.4 Send no Enabled Nodes
     - 2.1.5 Check if Req. # Operands Present
     - 2.1.6 Process Enabled Nodes
   - 2.2 Check if Done
     - 2.2.1 Test Node Count
     - 2.2.2 Write # to End_Flag
     - 2.2.3 Nodes are Queued
   - 2.3 Form Operation Packets
     - 2.3.1 Read Instruction Components
     - 2.3.2 Extract Needed Components
     - 2.3.3 Clean Enabled Node
   - 2.4 Read Result Packet
     - 2.4.1 Read Result Packet
     - 2.4.2 Place Value in Proper Instruction
   - 2.5 Blank Out Result Pack.
     - 2.5.1 Write Blanks to Packet Just Used
     - 2.5.2 Read Next Packet
     - 2.5.3 Test if 1st Char is Blank
   - 2.6 Dump Output Buffer
3. Slave Processor
   - 3.1 Check if End of Prog
   - 3.2 Jump to Location 0
   - 3.3 Read in Operation Packet
   - 3.4 Perform Operation
   - 3.5 Blank Out Packet Just Used
   - 3.6 Form Result Packet
   - 3.7 Write Result Packet

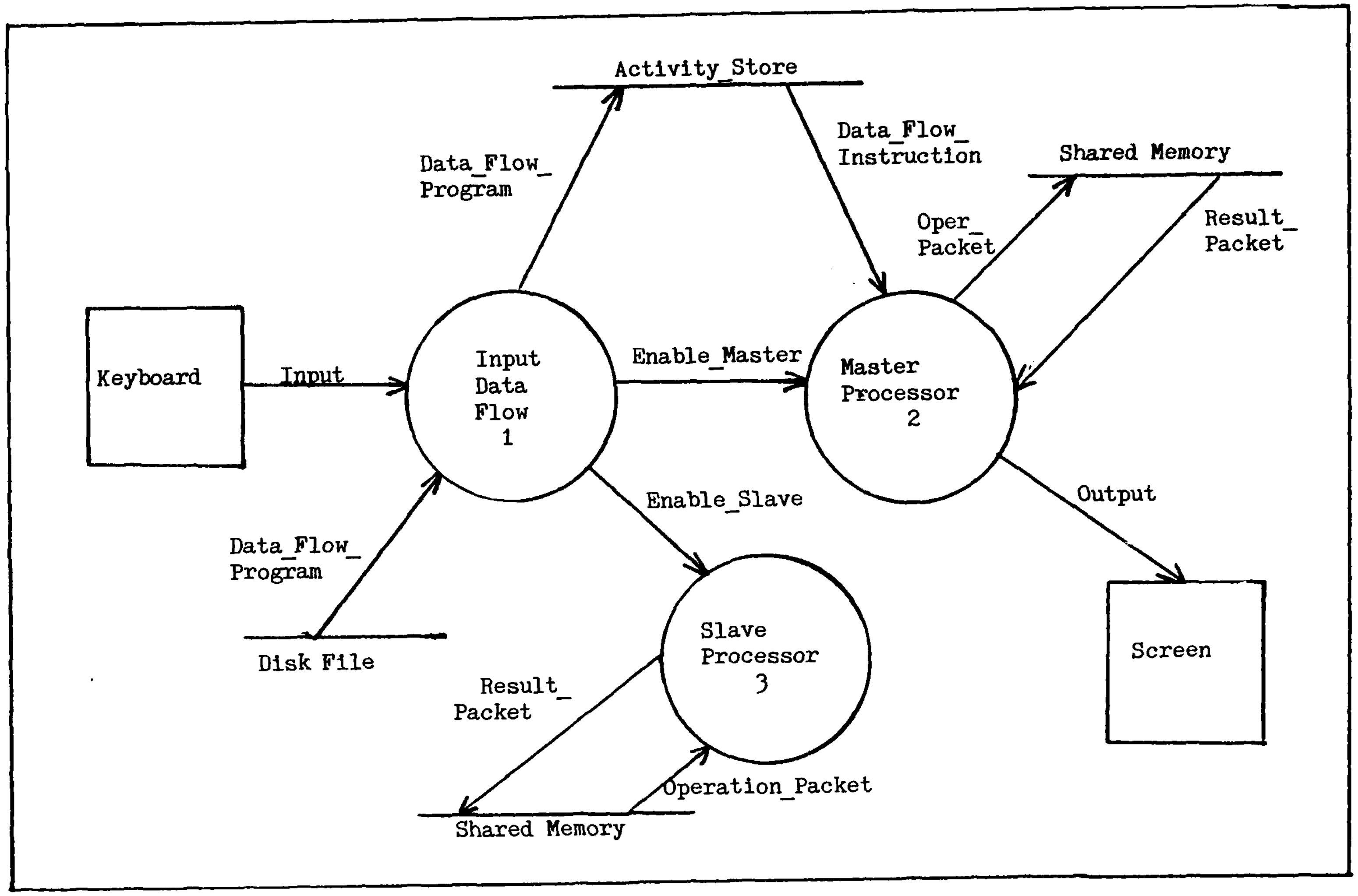


Fig. 10 DFD of Multiprocessor Software

shared memory. Process 1 then enables processes 2 and 3. Process 2 then reads the data flow instructions from shared memory (and processes them). Process 3 also reads instructions from and writes results to shared memory. When all instructions are executed, process 3 passes the output to the CRT screen. Obviously, this diagram is a simplified high level diagram of the function of the data flow processor. Each process can be further subdivided into other processes which explain the functions in more detail. The following paragraphs expand the DFD's and explain their functions.

<u>Input Process Requirements.</u> Figure 11 shows the second level data flow diagram expansion of the Input Process (1) of Figure 10. First, blanks must be written to the portion of shared memory set aside for storage of the activity templates (instructions), operation packets, and result packets. This ensures that previous RAM contents cannot accidentally interface with the processor software and cause erroneous results. This function is represented by process 1.1. Next, the data flow program must be read from a disk file. Process 1.2 reads in the program, and at the same time passes enable flags to enable the Master and Slave processors. However, before the processors can be enabled, process 1.3 must place the data flow instructions into activity templates and store them in the activity store. Once the activity store is filled, process 1.4 must load the Master and Slave processor software, and enable them. These

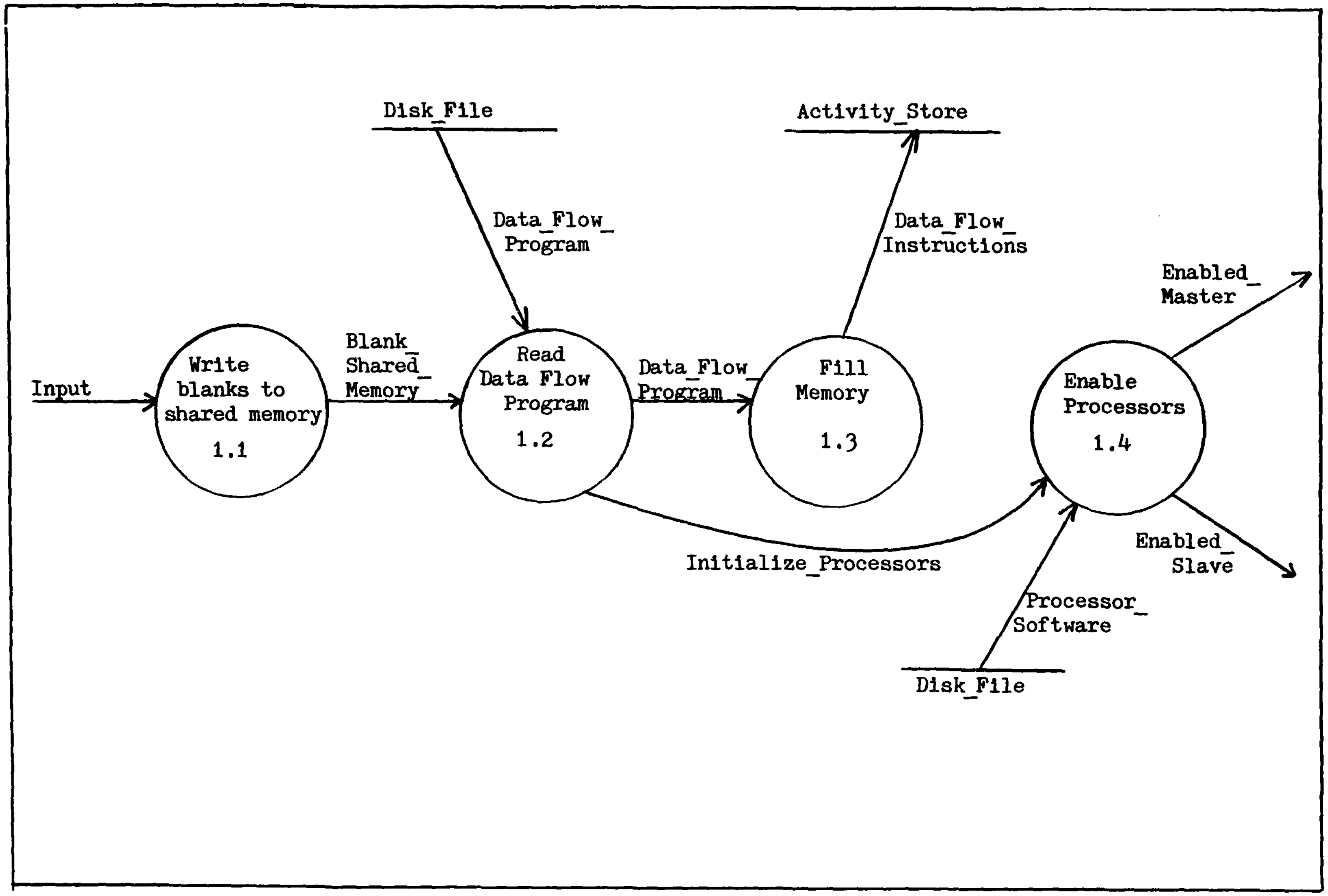


Fig. 11 Input Data Flow(1) DFD

processes can also be expanded to show greater detail.

Figure 12 shows the expansion of processes 1.3 and 1.4. The function of process 1.3 is to break the data flow program into data flow instructions and store them in the activity store. Process 1.3.1 must accept a program and break each instruction into its components. The components are the type of instruction (opcode), the operand values, the destinations, and the acknowledge signal information. To conserve storage space, the opcodes and acknowledge signal values are to require only one byte of storage for each value. The operand values are to be stored as word values. This will allow numbers greater than 255 to be computed. Process 1.3.2 converts the input operands into 2-byte values. The destinations are to be addresses which will point to where the result values are to be stored. Therefore, two bytes will be required to represent each address. Process 1.3.3 accepts the node numbers where the results will be sent, and uses them to compute the addresses. Process 1.3.4 stores all the components of the instruction in the activity store.

Master Processor Requirements. Figure 13 shows the second level expansion of the Master Processor (process 2) of Figure 10. This DFD shows that first the processor must check the activity store for enabled nodes (process 2.1). If enabled nodes are found, they must be converted into operation packets (process 2.3). If no enabled nodes are found, the processor must decide if execution should be

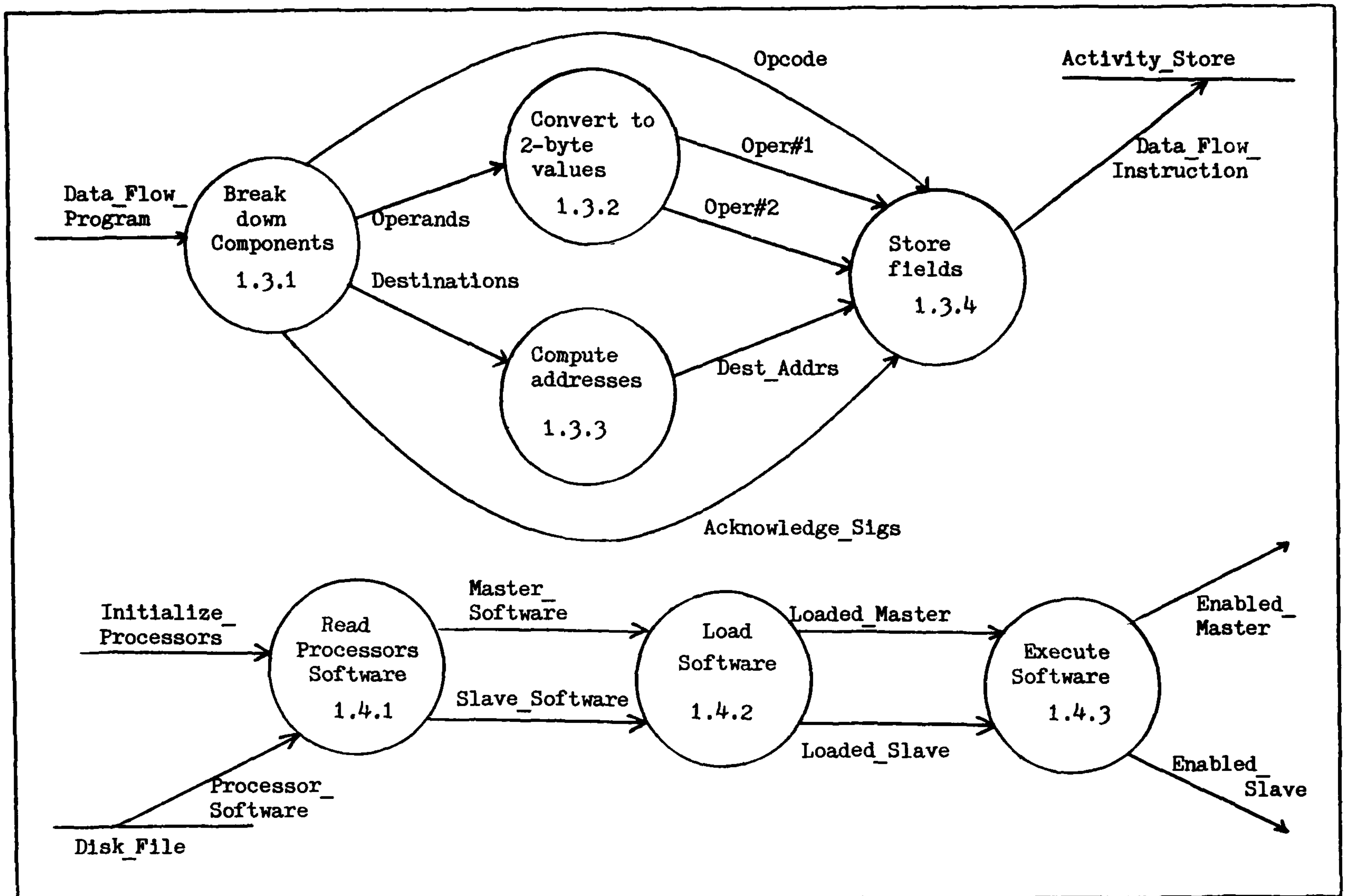


Fig. 12 Fill Memory(1.3) and Enable Processors(1.4) DFDs

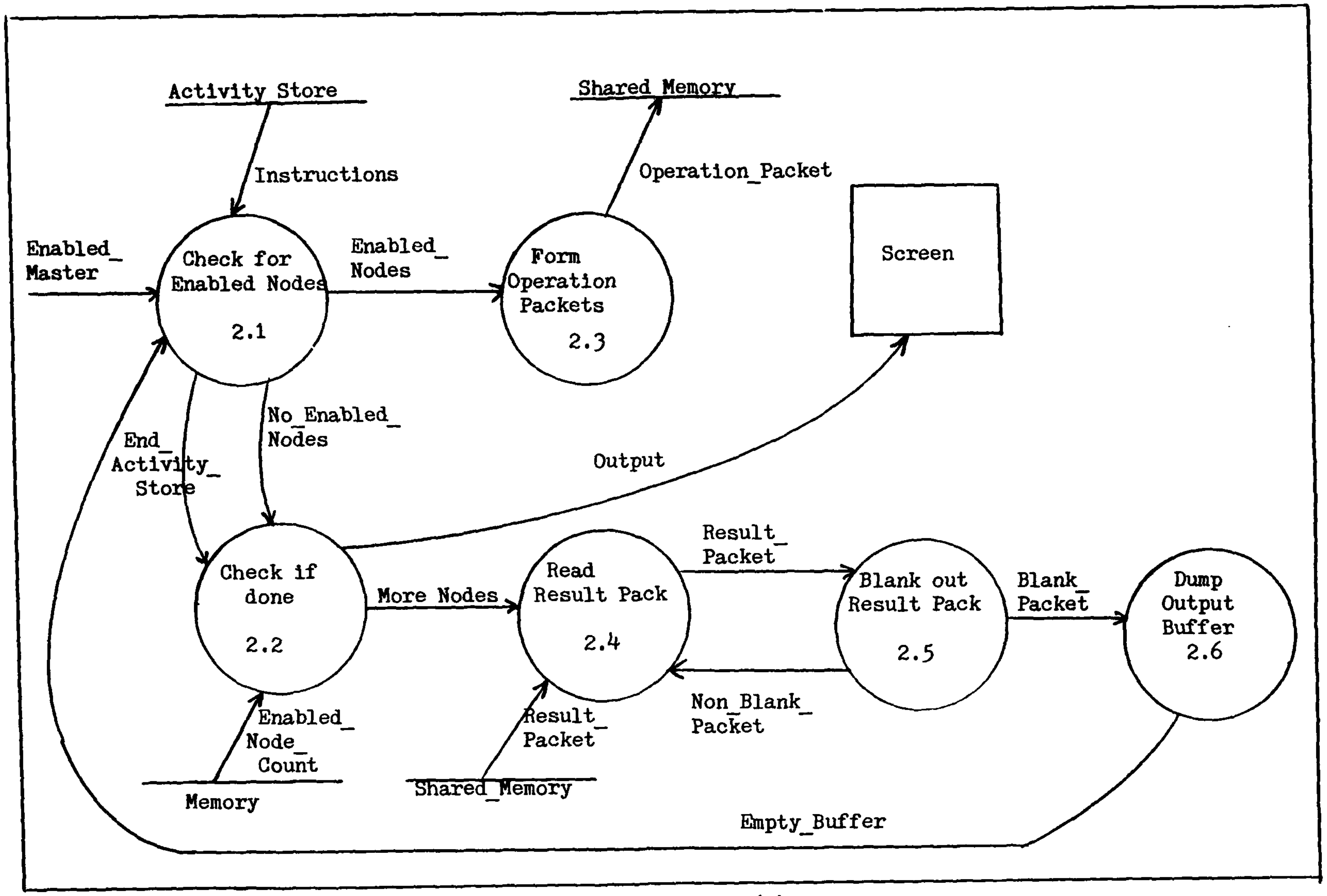


Fig. 13 Master Processor(2) DFD

stopped. This is done in process 2.2 by checking the enabled node count value. If that variable is greater than zero, enabled nodes have been processed and appear as result packets. Therefore, the result packets must be read (process 2.4), and processed (process 2.5). If the enabled node count variable is zero, execution ceases. Process 2.6 dumps the output buffer, if anything is in it, and sends an empty buffer to process 2.1. The program is repeated in this manner until the enabled node count variable becomes zero. Since these processes are fairly complicated, most will be expanded further.

Figure 14 shows the third level DFD expansion of process 2.1. This DFD shows that an instruction is read (2.1.1), and checked (2.1.2) if its acknowledge signals received parameter equals its acknowledge signals required parameter. If the signal parameters are not equal, the node cannot be enabled. That instruction must be marked as not enabled (2.1.4). If the signal parameters are equal, the operand fields must be checked to see if the operands are present (2.1.5). If the field contains hex "FFFF", the operand is empty and the node must be marked as not enabled (2.1.4). If the fields contain the required number of operands, the instruction is marked enabled (2.1.6). The address pointer must be incremented to point to the next instruction. The next instruction must then be checked to see if it is blank (2.1.3). If it is blank, then the end of the activity store has been reached. If the instruction is

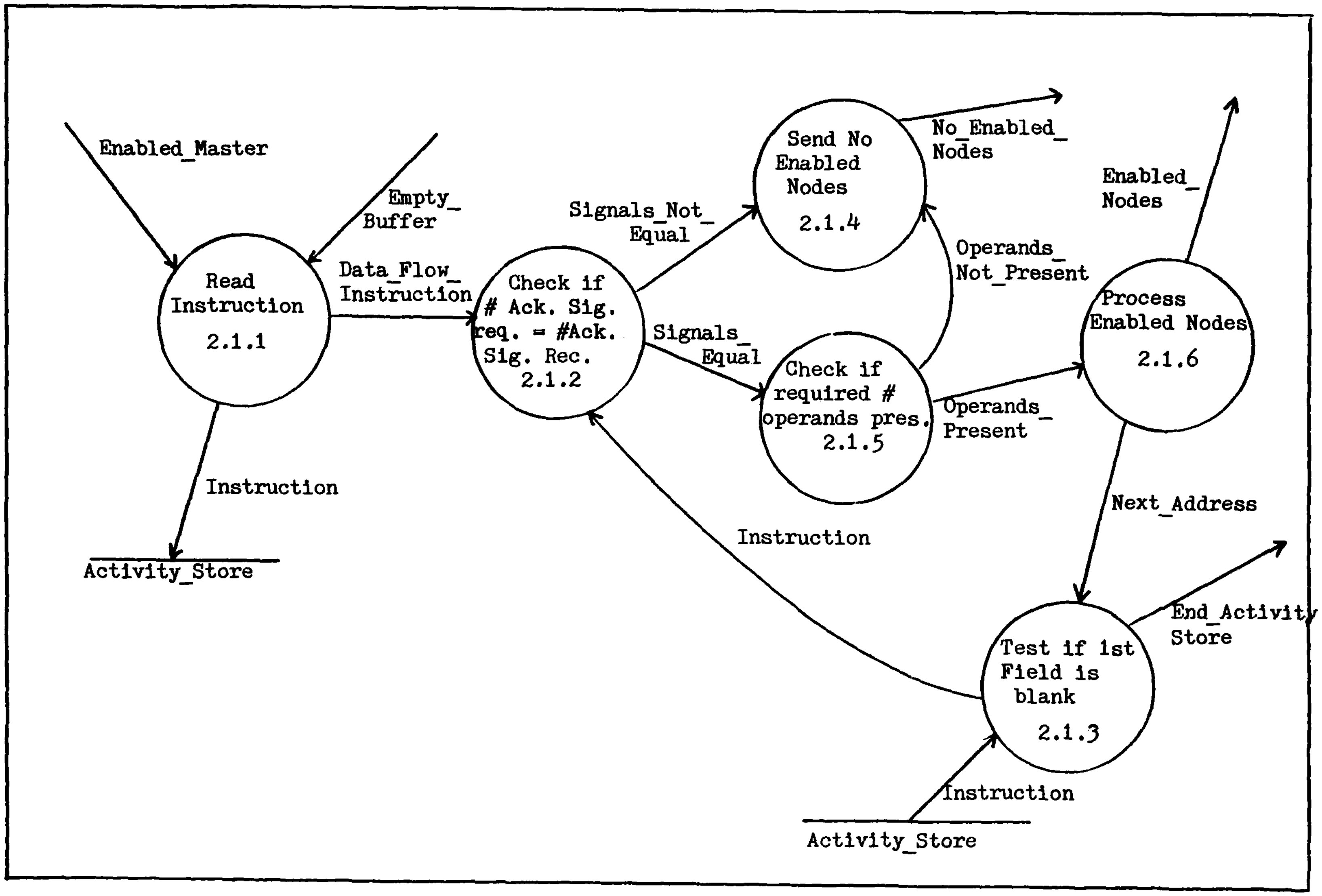


Fig. 14 Check for Enabled Nodes(2.1) DFD

non-blank, it must be checked to see if it can be enabled (2.1.2).

Figure 15 shows the expansion of processes 2.2 and 2.3 of Figure 13. Process 2.2 checks to see if the program has completed. Figure 15 shows that this function is accomplished by first checking the enabled node count variable. If the variable is zero, the program has completed. The slaves must also be told to quit. Therefore, the value "#" must be written to End_Flag which is a location in shared memory, and the Master must cease execution and pass its output to the CRT (2.2.2). If the enabled node count variable is not zero, program execution continues since the enabled nodes are queued in the result packets (2.2.3). Once the result packets have been processed, more nodes should be enabled. Figure 15 also shows how the operation packets must be formed. The enabled nodes are broken into the components of a data flow instruction (2.3.1). The components of an operation packet are extracted, and the packet is stored in shared memory (2.3.2). The enabled instruction must then be "cleaned up" (2.3.3). This means that the number of acknowledge signals received must be reset to zero, and the non-constant operands must be removed. If this step is not performed, the same node can be enabled an infinite number of times. After the "cleaning" step, the clean instruction must be stored back in the activity store.

Figure 16 shows the expansion of processes 2.4 and 2.5

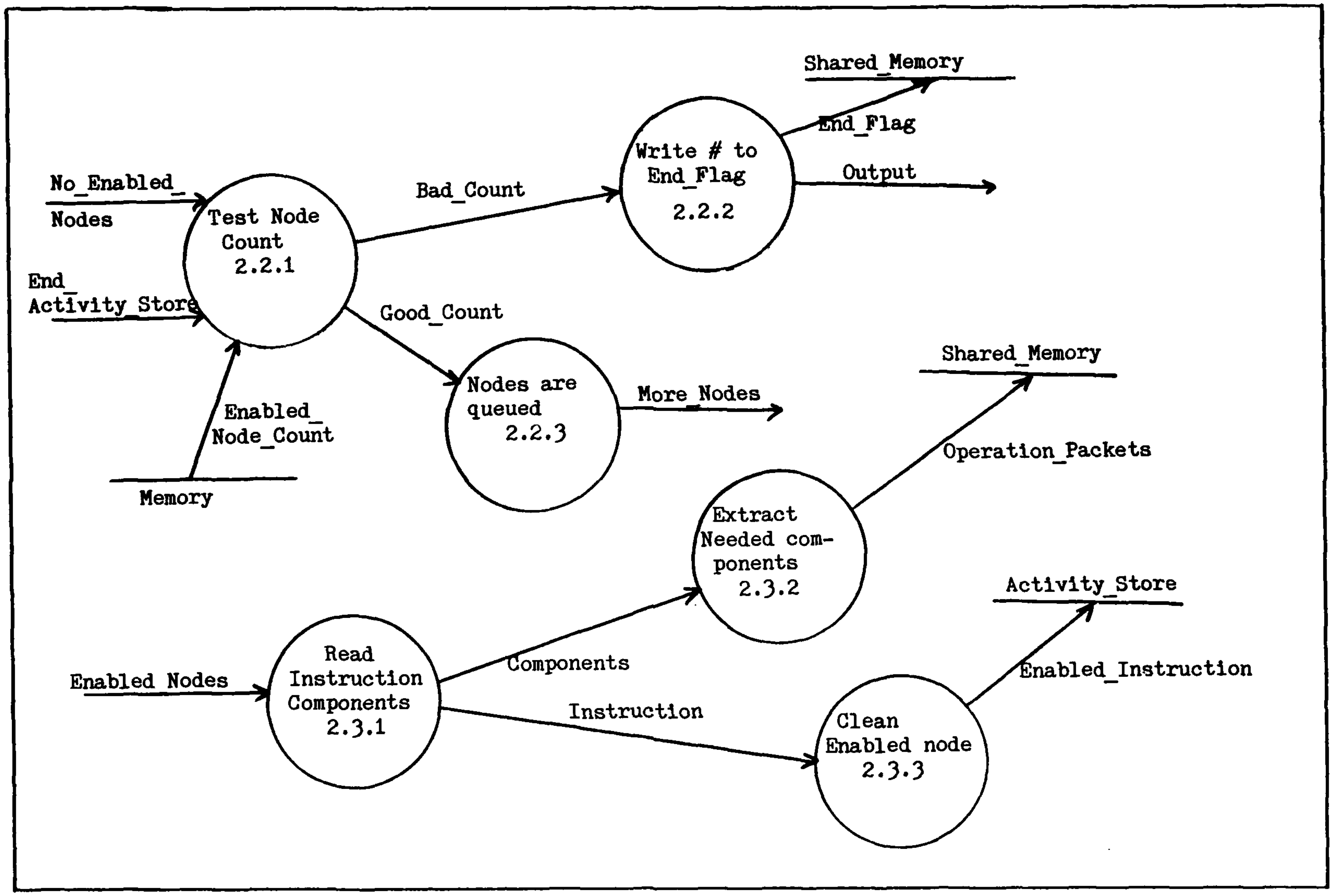


Fig. 15 Check if Done(2.2) and Form Operation Packets(2.3) DFDs

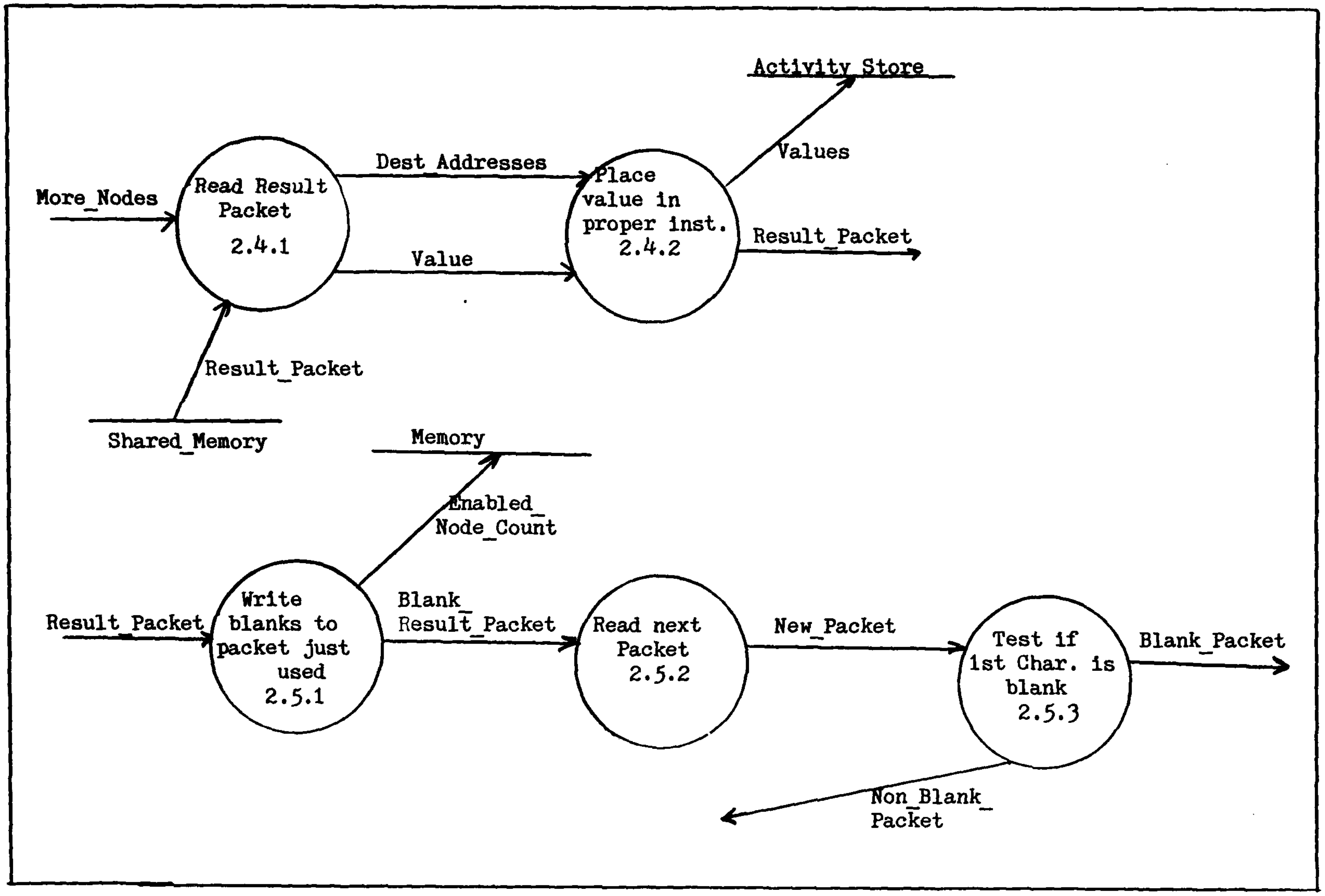


Fig. 16 Read Result Packets(2.4) and Blank Out Result Packets(2.5)

of Figure 13. Process 2.4 reads and processes the result packets. First, the result packet is read from shared memory (2.4.1). The destination addresses point to the instructions (locations) in the activity store where the result value is to be placed. Process 2.4.2 writes the result value into the proper instructions in the activity store. The result packet is then passed to process 2.5. This process writes blanks to the result packet just used (2.5.1), and decrements the enabled node count. The enabled node count variable can only be decremented by this process and is only decremented when a result packet has been processed. This is to ensure that all result packets will be processed before the enabled node count value equals zero (program termination). Process 2.5.2 reads a new result packet. If that packet is blank then all packets have been processed and the output buffer is dumped (2.6). If the packet is non-blank, it must be processed (2.4).

Slave Processor Requirements. Figure 17 shows the second level expansion of the Slave Processor (process 3) of Figure 10. This is the software which actually performs the data flow computations. Figure 17 shows a looping program that must first check if it should cease execution. Process 3.1 checks the contents of End_Flag. If a "#" is in End_Flag, the processor ceases execution (3.2) and jumps to location 0 (ROM). If no "#" appears in End_Flag, an operation packet is read in (3.3) from shared memory. Next, the operation specified in the operation packet must be

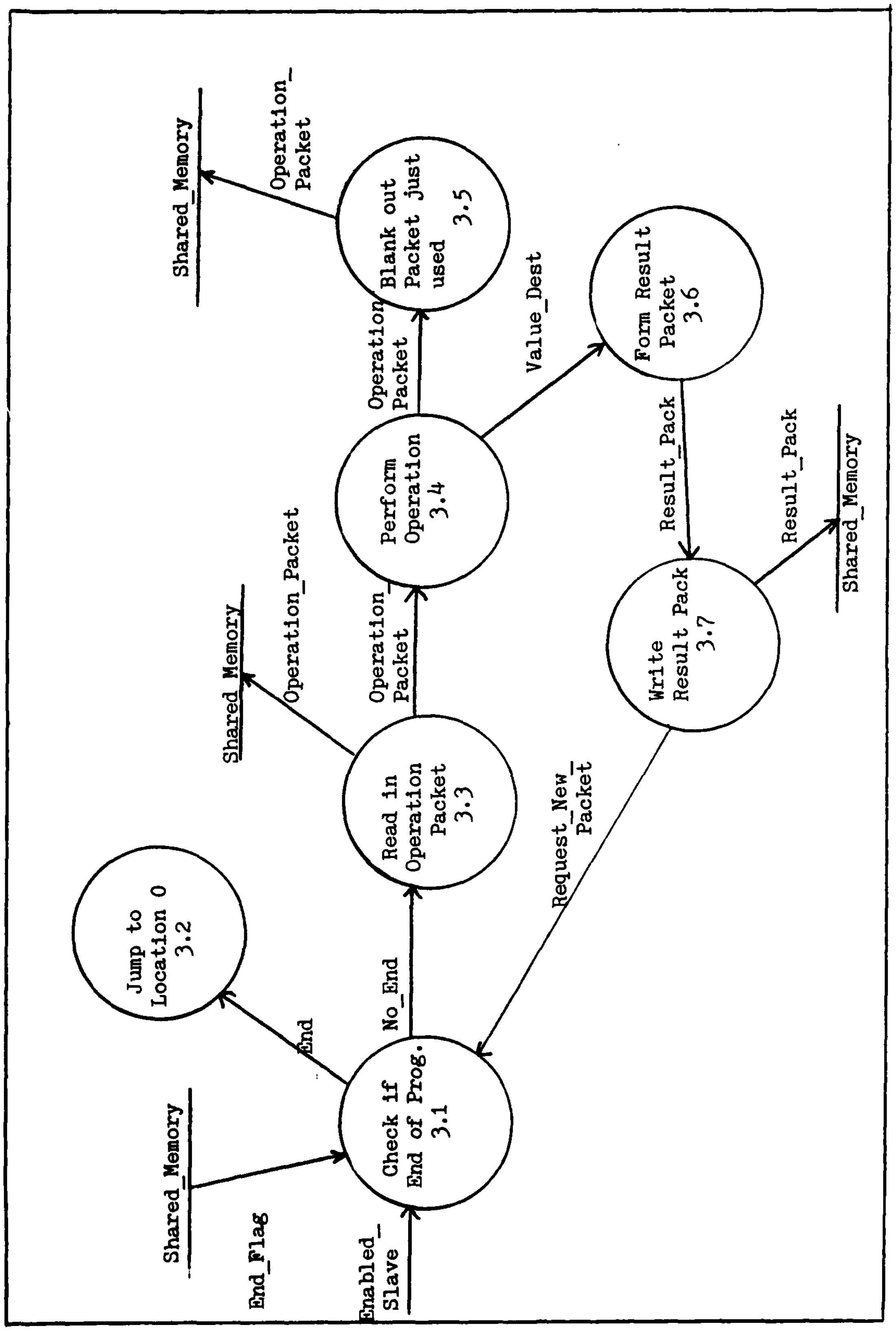


Fig. 17 Slave Processor (3) DFD

performed (3.4). Types of operations must include: addition, subtraction, multiplication, division, operand comparisions, and exponentiation. This process must also extract the destination addresses from the operation packet. Next, the result of the operation and the destination addresses are combined to form the result packet (3.6). At the same time, the operation packet must be blanked out so that space will be available for future operation packets (3.5). The result packet from 3.6 must be stored in shared memory so that the Master Processor can access it (3.7). Finally, a request for a new operation packet is sent which initiates the loop again.

Summary

This chapter has defined the requirements for both the software and the hardware of the data flow multiprocessor. Additionally, structured analysis techniques were introduced and illustrated. These requirements form the basic foundation for the overall design of the multiprocessor system described in Chapter V.

V. Design

Introduction

This chapter translates the requirements specified in Chapter IV into an actual design of a data flow processor. The first section of the chapter presents the design for the data flow language in terms of its representation and its function. The following sections present the design of the hardware and specifies how the master processor communicates with the slave processor(s). Next, the partitioning of the RAM memory is discussed and is illustrated by providing a memory map. The remaining two sections of the chapter present the design of the software which implements the functions of the master and slave processor(s) respectively. The software for these processors was designed by using structured design techniques to translate the structured specification (from the previous chapter) into structure charts.

Data Flow Language

The data flow language is the vehicle used to represent data flow instructions to the multiprocessor, and is the source language of the multiprocessor. A particular data flow language is designed for use only on the multiprocessor described in this thesis. Because of this, the language is similiar to the languages in the literature, but contains some differences not found in other languages. These differences are not capabilities added, but rather, they are capabilities not present. For example, this language does not support data structures such as linked lists, and stacks, and does not support character or text processing. Also, the language allows only binary operations, rather than multi-valued operations. The reason that these capabilities are not present is to make the design and implementation of the language as uncomplicated as possible. The objective of this investigation is to design and implement a data flow multiprocessor. Future investigations can modify the language to include these capabilities. The following paragraphs discuss other reasons for excluding these capabilities.

The reason that only binary operations are allowed is to make the storage of the data flow instructions as efficient as possible. The term binary operation, as used here, means that the operation can have at most two operands, and can only pass the result of the operation to at most two other operation nodes. This allows each data flow instruction to be represented, and stored, in only 16 bytes of RAM. As will be explained later, this 16 bytes includes all operands, destination addresses, and acknowledge signal information required to process a single data flow instruction. If the instructions were allowed any number of input operands, or allowed to pass results to any number of operations, the amount of storage required to store an instruction would differ from instruction to instruction. Linked lists would be required to store all input operands and output destinations. This would be messy and require a large amount of memory. With this method of representing an instruction, 16 contiguous bytes of storage will be required, as opposed to having instruction operands stored throughout memory.

Data structures such as stacks and linked lists are not represented by this language for two reasons. First, the amount of memory required to represent these data structures would be prohibitive. The second reason is that of complexity. The representation of stacks and linked lists would require contiguous memory allocated for stacks and an extra instruction value to hold pointers. This makes an already complex task even more difficult, if not impossible, due to the RAM memory limitations. Future thesis efforts could take up the issue of representation of data structures and character manipulation.

The data flow language is graphically represented as nodes and arcs, as in Figure 1. Each node represents one binary operator, and each arc represents movement of data values through the operations. A data flow program is represented by a combination of nodes and arcs, and has the appearance of a binary tree. The data flow language is represented to the multiprocessor as a 16 byte value containing operator type, operands, destinations, and acknowledge signals. The following sections define the types of operators, and the 16 byte storage values.

Operators. The data flow language can represent 14 different operations. These operations are defined in Table II. There are four arithmetic operators which perform the specified operation on the two inputs, and pass the result to the output destinations. There are four comparision operations which compare one input to the other and pass either the true (T) or the false (F) boolean values to a single output, depending upon the result of the comparision. There are six specialized operations which handle input, output, exponentiation, replication, and switching depending on the boolean input value. The following paragraphs discuss in detail the function of each operator.

The arithmetic operators perform the specified operation on the two input values. The division and subtraction operators need further explanation. Unlike the other two arithmetic operations, they are operand-sensitive. This means that the location of input operands are critical to the successful completion of the operation. For example, the following computation,

$$6 - 5 = 1,$$

does not produce the same result as,

$$5 - 6 = (-1).$$

In this example the two input operands are the same, but their locations were switched. To prevent this ambiguous computation present in both subtraction and division

Table II

Operators Defined in the Multiprocessor

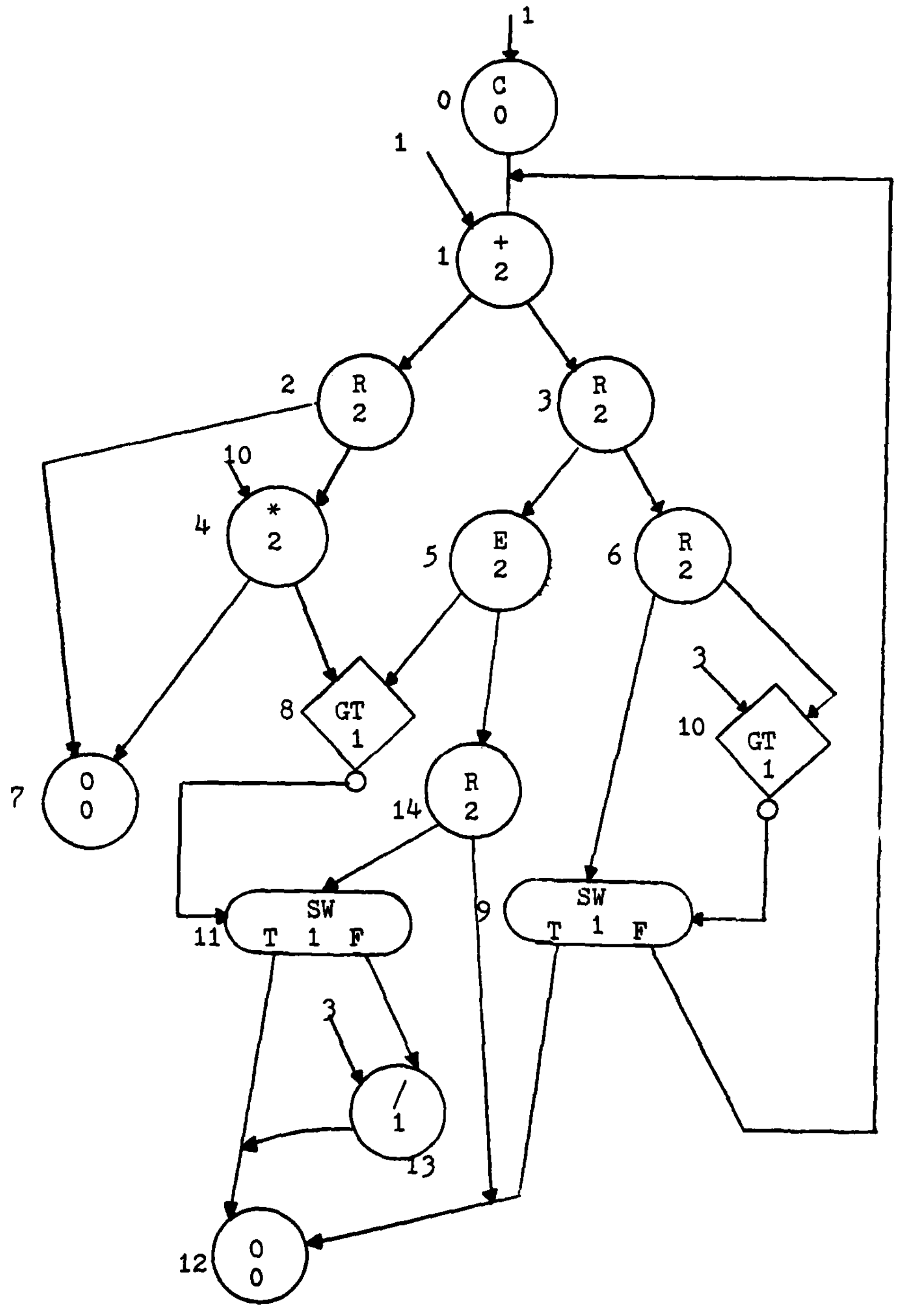
Operation	Name	Definition
+	add	takes two inputs and performs the operation passing the result to all operands.
-	subtract	
*	multiplication	
/	divide	
>	greater than	compares input operand #2 to input operand #1, if true passes T boolean value, if false passes F value to single output.
<	less than	
#	not equal	
=	equal	
C	literal	passes single constant to all outputs.
E	exponentiation	squares input and passes output.
R	replicate	passes input value to all outputs.
I	input	read single value from console, pass to all outputs.
O	output	output all input operands to list device.
S	switch	passes numeric input value to T or F branch depending on input boolean value.

operations, the second operand will be the value subtracted from or divided by the first operand. For example, if operand #1 was 6 and operand #2 was 18, the subtraction operation would produce the value 12 and the division operation would produce the value 3.

The comparison operations all perform the specified comparison on the two inputs and produce a boolean value as a single output. The output is either a "T" or "F" depending on the result of the computation being either true or false respectively. The output result is passed to a switch operation, which branches to either the true path or the false path, depending on the boolean input value. The less-than and greater-than operations are operand-sensitive. Again, ambiguous results could be produced by these twc operations. To prevent this, operand #2 will always be compared to operand #1. For example, if operand #1 was 10 and operand #2 was 20, the greater-than operation would produce a "T" result, and the less-than operation would produce a "F" result.

The specialized operations must be explained individually. The "C" operator is a unary operation where the single constant input value is passed to the output destinations. The replicate, or the "R" operation, is another unary operation where the single input value (constant or variable) is passed to all outputs. The exponentiation operator takes the single input, squares it, and passes the result to all outputs. The input and output operations read from and output to the console device. In the case of the input operation, the value input is passed to all output nodes. In the case of the output operation, all input operands are displayed on the console device. This operation has no output arcs and thus passes no value to any other nodes. The final specialized operation is the switch operator. This operator has two types of inputs. Operand #2 is a boolean value (T or F) passed from any one of the four comparison operations. The switch operation passes the operand #1 value to either output #1 (if the boolean input was true), or to output #2 (if the boolean input was false). Output #1 is always the true branch, and output #2 is always the false branch.

Representation. As already seen, the data flow language is graphically represented by means of nodes and arcs. It is logically represented to the multiprocessor by specifying operator type, operands, destinations, and acknowledge signals. All of these values must be expressed for each instruction. Figure 18 shows a graphical representation of a simple PASCAL program defined in Figure 19. Figure 20 shows how this program is logically represented. Each instruction must contain values for the following twelve fields. Field #1 is the node number. It is given an integer value from 0 to 255. The first instruction must always be node #0. Field #2 is the one character designation for the operation type. There are 14



values inside nodes = # acknowledge signals required to fire
values outside nodes = node number

Fig. 18 Sample Data Flow Program Representation

```
                FOR I := 2 TO 3 DO
                     BEGIN

                     J := I * 10;

                     K := I * I;

                     IF K > J THEN L := K

                          ELSE  L := K DIV 3;

                     WRITELN (I,J);
                     WRITELN (K,L);
                     END;
```

Fig. 19 Sample PASCAL Program

0	C	1	1	1	0	1	2	0	0	0	0
1	+	2	1	1	2	2	1	3	1	0	0
2	R	1	0	0	2	7	1	4	2	1	0
3	R	1	0	0	2	5	1	6	1	1	0
4	*	2	1	10	2	7	2	8	1	2	0
5	E	1	0	0	2	8	2	14	1	3	0
6	R	1	0	0	2	9	1	10	2	3	0
7	O	2	0	0	0	0	0	0	0	2	4
8	>	2	0	0	1	11	2	0	0	4	5
9	S	2	0	0	1	12	2	1	2	6	10
10	>	2	1	3	1	9	2	0	0	6	0
11	S	2	0	0	1	12	1	13	2	14	8
12	O	2	0	0	0	0	0	0	0	13	14
13	/	2	1	3	1	12	1	0	0	0	0
14	R	1	0	0	2	11	1	12	2	5	0

Fig. 20 Logical Representation of PASCAL Program

possible values corresponding to the 14 operands in Table II. Field #3 is the number of inputs to the node. The possible values are 1 or 2 (integer). Field #4 is the number of constants in the instruction. There can be at most one constant in any node. Therefore, the possible values are 0 or 1 (integer). At the completion of execution of an enabled node, the operands are removed to prepare the node for the arrival of the next set of operands. Field #4 is required to tell the software to remove only the non-constant operands. This means that constants are not removed and then reloaded. Field #5 is the constant value (if any). Its possible value can be any integer value which can be expressed in a 16-bit word. If a constant value is present, it will always be placed in operand #1 of the instruction. The two operands are both 16-bit words. Field #6 is the number of acknowledge signals required for the instruction to fire. The possible values are 0, 1, or 2 (integer). Field #7,8,9,10 are the destination fields. Field #7 and 9 are the node numbers which the result of the computation is passed to. Field #8 and 10 are the input numbers of that node. All four of these values are 8-bit integers. Finally, Field #11 and 12 are the node numbers which the acknowledge signals must be passed to. They are also 8-bit integers. The multiprocessor software takes this data flow instruction representation, and converts it into the 16 byte storage format used to actually process the instruction.

Hardware Design

The hardware chosen for this design will consist of 64k RAM memory, a CRT, a floppy disk system and three microprocessors (one processor serves as the master and two serve as slaves). The microprocessors chosen are two Intel SBC 80/20's and one SBC 80/30. There are a number of reasons that these processors were chosen. They support multiprocessor access to a common memory, they support interlock functions, they were available for use, and they were used in the previous thesis investigation.

The multiprocessor system implemented in this thesis will use a master/slave relationship. In this case, the master processor will process all input and output and will enable the slave processors. The slave processors can only be initiated by the master and they will process the actual data flow instructions. Figure 21 shows a block diagram of how the hardware is designed. The master processor is the only processor which can communicate with the input/output devices (CRT and disk system). It also communicates to the slave processors via shared memory. The slave processors communicate directly with the shared memory. They are enabled by the master processor when a certain value is written into a specific shared memory location. This is explained in the next section.

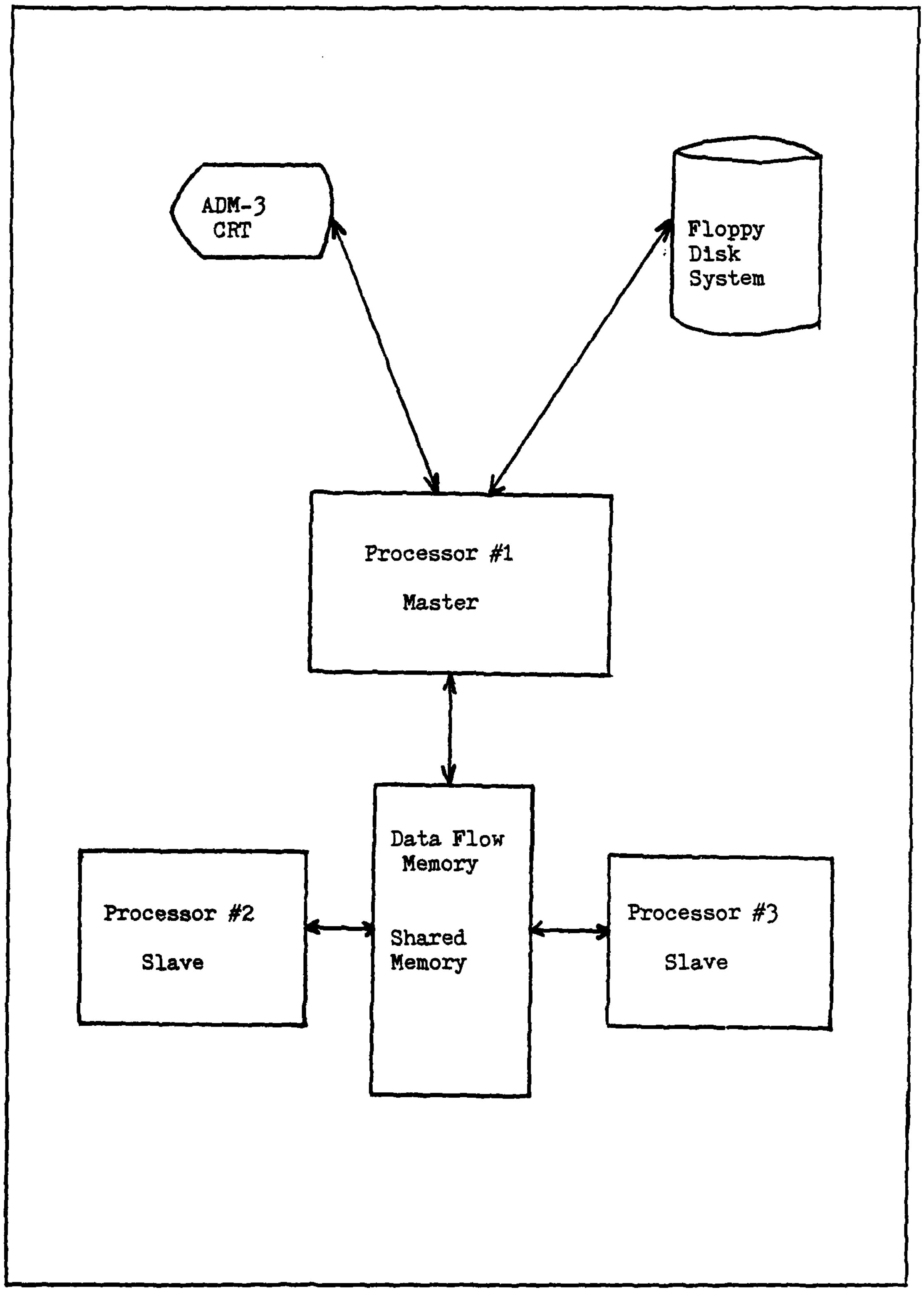


Fig. 21 Hardware Block Diagram

Inter-processor Communication

The master and slave processors communicate via shared memory. The master enables the slave by writing a value of 1 into memory location F7FCh (the "h" means hexidecimal). When a value 1 appears in F7FCh, the slave reads the address of the task to be executed at location F7FDh. It sets location F7FCh to zero and jumps to the address it just read (in F7FDh). When the task has completed, the processor jumps to location 0h, and waits to be assigned again. A ROM program, which executes this process is displayed in Figure 22 (Ref 6). It is placed in the slave processors' ROM, replacing the ROM monitor program. This program executes continuously in the slave processors, which means that the slave processors do not process any data flow instruction until commanded by the master via location F7FCh in shared memory.

Shared Memory

Shared memory does much more than serve as the communication medium for the processors. The 64k memory board, which all processors have access to, also stores the data flow processor software, and the actual data flow program in the activity store. A memory map of the entire 64k memory is shown in Table III. Note that most of the memory is common to both processors, but there are some areas that belong to one or the other processor alone. Specifically, each processor has its own RAM memory from

```
           ROM        .EQU      0
           ASSIGN     .EQU      0F7FDH    ;WHERE ASSIGNMENT
                                          ;WILL BE
           LOC        .EQU      0F7FEH    ;WHERE JUMP ADDR IS
                      .ORG      ROM
                      LD        A,128
LOOP:
                      DEC       A
                      JP        NZ,LOOP
                      LD        A,(ASSIGN)
                      CP        1
                      JP        Z,GOTONE
                      LD        A,10
                      JP        LOOP
GOTONE:
                      LD        HL,(LOC)
                      XOR       A
                      LD        (ASSIGN),A
                      JP        (HL)
                      .END
```

Fig. 22 ROM Program in Slave Processors

Table III

Memory Map of 64k Shared Memory

Address	Master	Slaves
0000 - 0FFF	Monitor Rom	Rom Program
1000 - CFFF	PASCAL System	not used
D000 - D7FF	not used	not used
D800 - DFFF	Shared Output Buffer for Both	
E000 - E7FF	Activity Store for Both	
E800 - EFFF	Communication Packet Storage	
F000	END_FLAG word location	
F001 - F7FB	not used	not used
F7FC - F7FF	Shared Processor Assignment Addr	
F800 - FFFF	Master Dataflow Processor Code	Slave Data-flow Process Code

location F800h to FFFFh. This is where the machine code of the data flow processor resides. Putting the code in the individual processors' RAM has two major advantages over using the common memory; it minimizes bus traffic, because accesses to internal RAM does not require access to the system bus; and it allows the processors' code to occupy the same address space without having to be reentrant code (Ref 6). The processors communicate via memory locations F7FCh to F7FFh. Another important partition of memory is locations E000h to F000h. Locations E000h to E7FFh are the activity store where each instruction is stored as a 16 byte entity. Figure 23 shows the structure of the activity store, described in the Circular Pipeline Section of Chapter 3. The word number values must be added to an address (the address must fall on a 16 byte boundary) to access a particular field. For example, address E00Dh would access the "number of acknowledge signals required to fire the node" field of the first instruction. Address E013h would access the "# of operands" field of the second instruction. Since only 2k has been reserved for the activity store, only 128 data flow instructions can be represented. The remainder of the shared data flow program memory (E800h-EFFFh) is used to store, and process, the communication packets. Figure 24 shows the structure of this 2k partition. Note that an operation packet comprises ten fields (words 0-9), and a result packet comprises six fields (words 9-F). The "node #" field is common to both packets.

Word Numbers

	0	1	2	3	4	5,6	7,8	9,A	B,C	D	E	F
	# Ack. Signals Rec.	Node		# Operand	# Constant	Operand #1	Operand #2	Dest #1 Address	Dest #2 Address	# Ack. Sig Req.	Trans. Ack. Sig Node #	Trans Ack. Sig Node #
		#	Type									
E000												
E010												
. . . ADDRESS												
E7F0												

E7FF

Fig. 23 Structure of the Activity Store

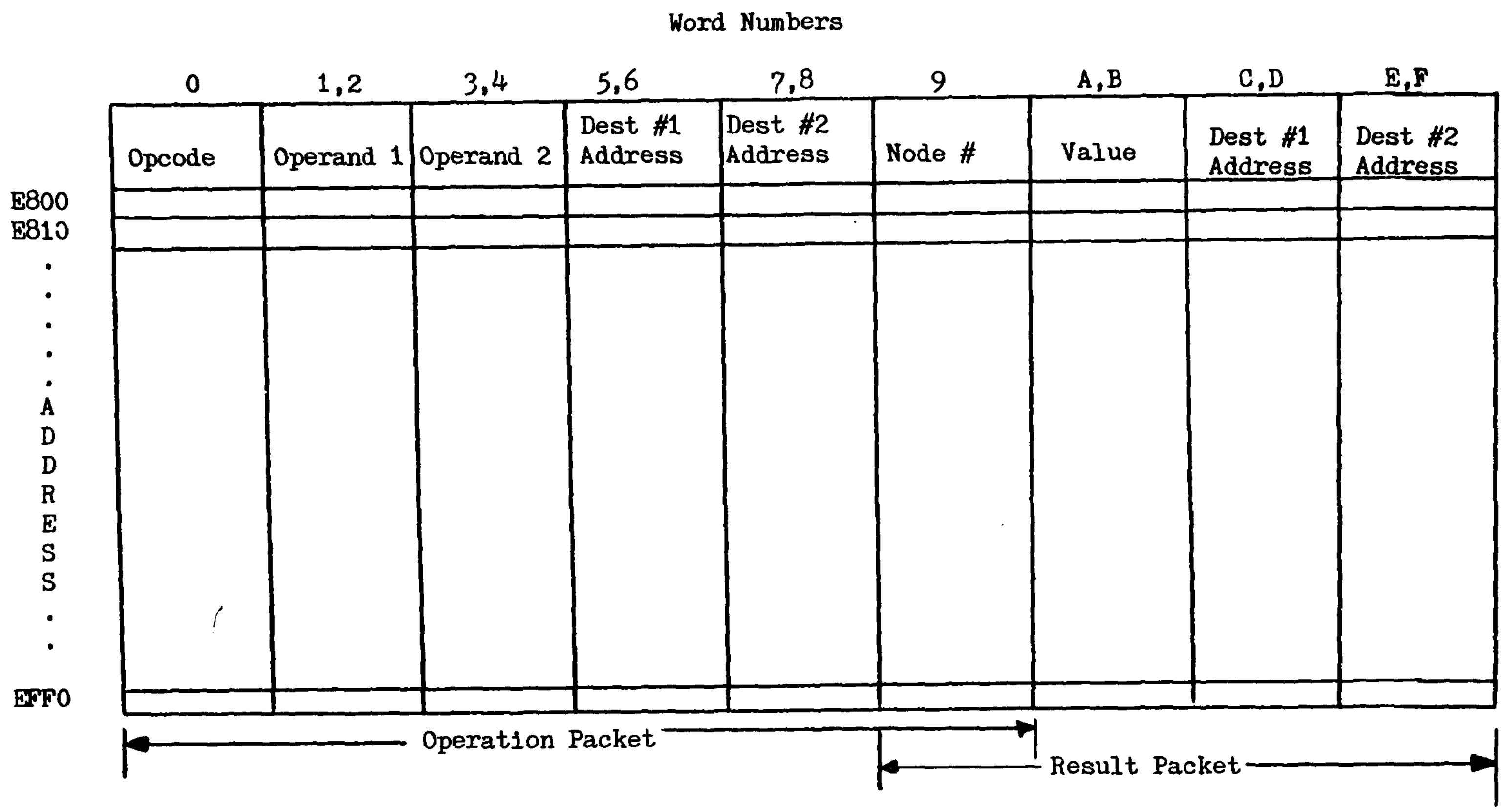


Fig. 24 Structure of the Communication Packets

There are 128 packets available. When the processor software determines that an instruction (activity template) has all required operands, it is converted into an operation packet. Once that has executed, its result is placed in the second field of the result packet. The software uses the result packet that was created and places the result in the proper addresses in the activity store. One final location has an important function. Location F000h is the "END_FLAG" field. When a "#" value is placed in this word by the master processor, the slaves cease execution and jump back into their loop programs. Only the master processor writes to this shared location.

Master Processor Software

The master processor software is written in assembly language and resides in locations F800h to FFFFh of the master processor. The master processor functions as both the fetch unit and the update unit as described in Figure 8. The function of the software can be stated in the following structured English format:

```
Begin  (* MASTER *)
     While there are Enabled Nodes DO
     Begin  (* WHILE *)
          Increment Enabled_Node_Count
          Lock Shared Memory
          Form Operation_Packet
          Clean up Enabled_Node
          Unlock Shared Memory
          Read next non-blank Result_Packet
          Lock Shared Memory
          Write Value to Destinations
          Blank out Result_Packet
          Decrement Enabled_Node_Count
          Unlock Shared Memory
          If character in Output Buffer
              Dump Buffer
     End  (* WHILE *)
     If no Enabled_Node
          Write # to END_FLAG
End  (* MASTER *)
```

The DFD which describes the function of the master processor software appears in Figure 13. Using structured design techniques the afferent (input) and the efferent (output) sections were identified, and the structure chart of Figure 25 was derived. This structure chart shows the software modules which comprise the master processor software.

The structured English description and the structure chart state that the master software searches through the activity store looking for an enabled node. A node is defined as enabled when the required number of operands are present (field #3, Figure 23) and the number of acknowledge signals received equals the number of acknowledge signals required to fire (field #0 and field #D). When an enabled node is found, shared memory is locked (to prevent interference from the slave), and an operation packet is formed in the first available 16 byte location of the communication packet partition. Next, the enabled node is cleaned up in the activity store. This means that all non-constant operands are removed, and the number of acknowledge signals received field is set to zero. This is done so that the node can be re-enabled when all of its operands arrive. The shared memory is unlocked so that the slaves can execute. Next, the master software reads through the 2k

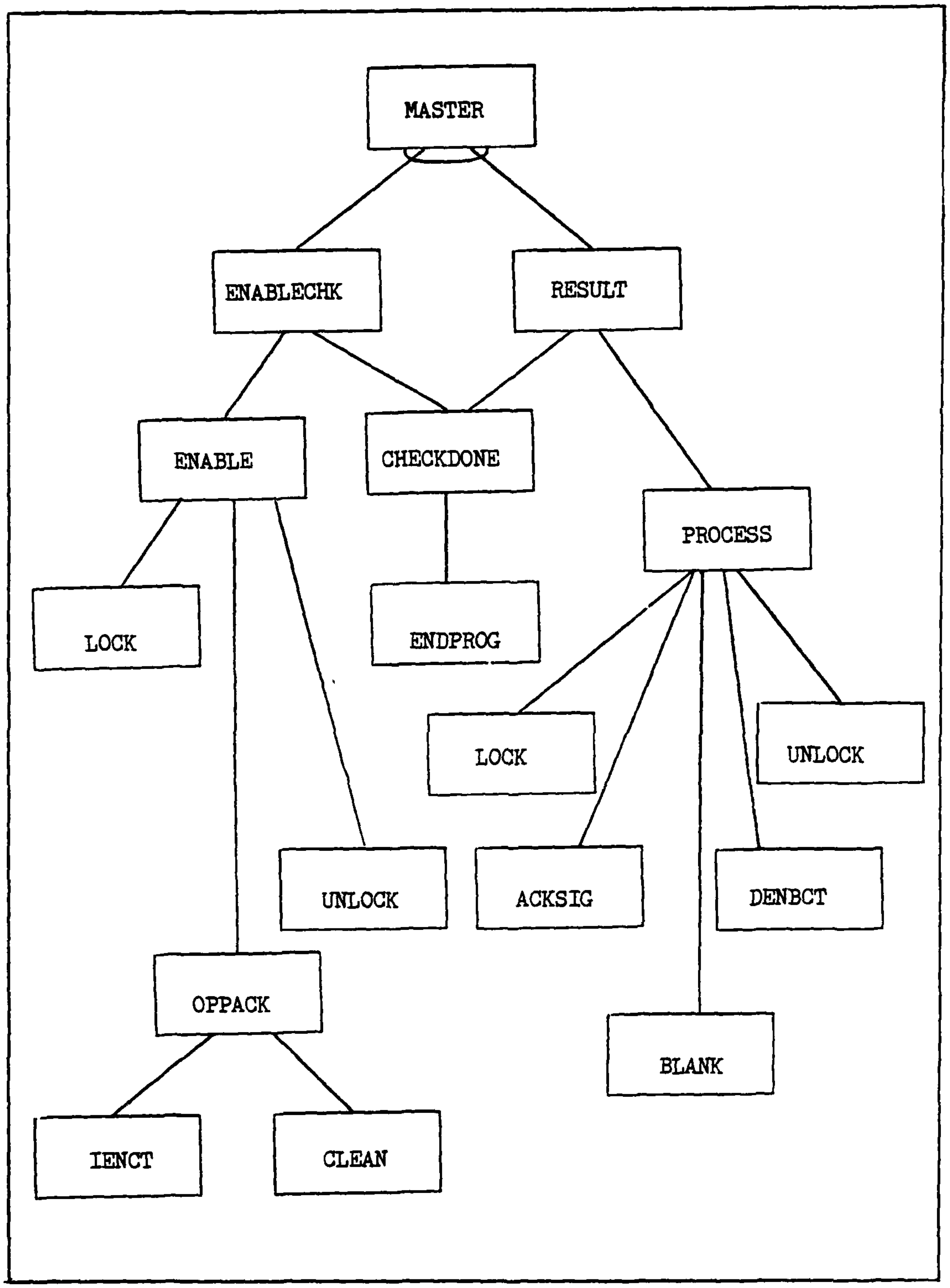


Fig. 25 Master Processor Structure Chart

communication packet partition, searching for the first non-blank result packet. When one is found, the memory is locked, the result value is sent to the appropriate destinations, blanks are written to the result packet just read, and the memory is unlocked. The final task performed in this loop is dumping the output buffer. The software continues to search for enabled nodes until none are found. When this happens, a "#" value is written to END_FLAG (location F000h). This signals the slaves to cease execution, and the data flow program terminates.

Slave Processor Software

The slave processor software is also written in assembly language and resides in locations F800h to FFFFh of the slave processor. The slave processor functions as the operation unit as seen in Figure 8. The function of the slave processor can be stated in the following structured English format:

```
Begin  (* SLAVE *)
     While END_FLAG not equal # DO
     Begin (* WHILE *)
          Read Operation_Packet
          Move to work area
          Perform operation
          Lock Shared Memory
          Blank out Operation_Packet
          Form Result_Packet
          Write Result_Packet
          Unlock Memory
     End (* WHILE *)
     Jump to location 0h
End (* SLAVE *)
```

The DFD which describes the function of the slave processor is shown in Figure 17. The structure chart which defines the modules of the slave processor software is shown in Figure 26.

The structured English description and the structure chart state that the slave processor software is composed of a continuous loop. The first thing that the program does is to check for the end of execution. This is done by checking the END_FLAG value. If a value "#" appears, execution stops. If any other value is in END_FLAG, the program reads in the first operation packet encountered. The appropriate values are moved to internal program locations and the specified operation is performed on the operands. Next, the memory is locked, and blanks are written to the operation packet just read (thus freeing this location for another operation packet). The result of the operation is now used to form a result packet which is written into shared memory. Finally, the memory is unlocked, and the program continues the loop.

Pascal Host Software

Before either of the two looping programs (master and slave) are executed, the Pascal host first writes blanks to memory locations E000h to F000h. This is to insure that the activity store and communication packet partitions are free of errant data values from some other program. The Pascal host software also reads in the data flow notation and writes it to the activity store. It also enables the slave processors. This is done before the master starts execution, so that the slave processes the very first operation packet made available by the master. The DFD diagram which depicts the function of the host software is shown in Figure 11. The structure chart which defines the modules of the host software is shown in Figure 27.

Alternative Designs

This chapter has presented just one design for a data flow multiprocessor. There were other design approaches available. For example, in one approach each node has its own processor (Ref 3). In another approach, all nodes draw on a central processor pool, and return the processors to the pool when completed. The second approach was the design implemented in the previous investigation (Ref 6). Both of these approaches could have been implemented in this investigation, but were not for various reasons.

The first approach was not chosen because of the number of processors required. Many researchers envision this type of machine to contain hundreds or even thousands of processors interconnected to form one supercomputer (Ref 3). With this many processors, a machine could theoretically achieve execution speeds in the billion instruction per second range (Ref 12). This type of machine is far beyond the scope of one MS thesis. Also, a limited number of microprocessors were available. To even implement the simple data flow program of Figure 18, fourteen processors

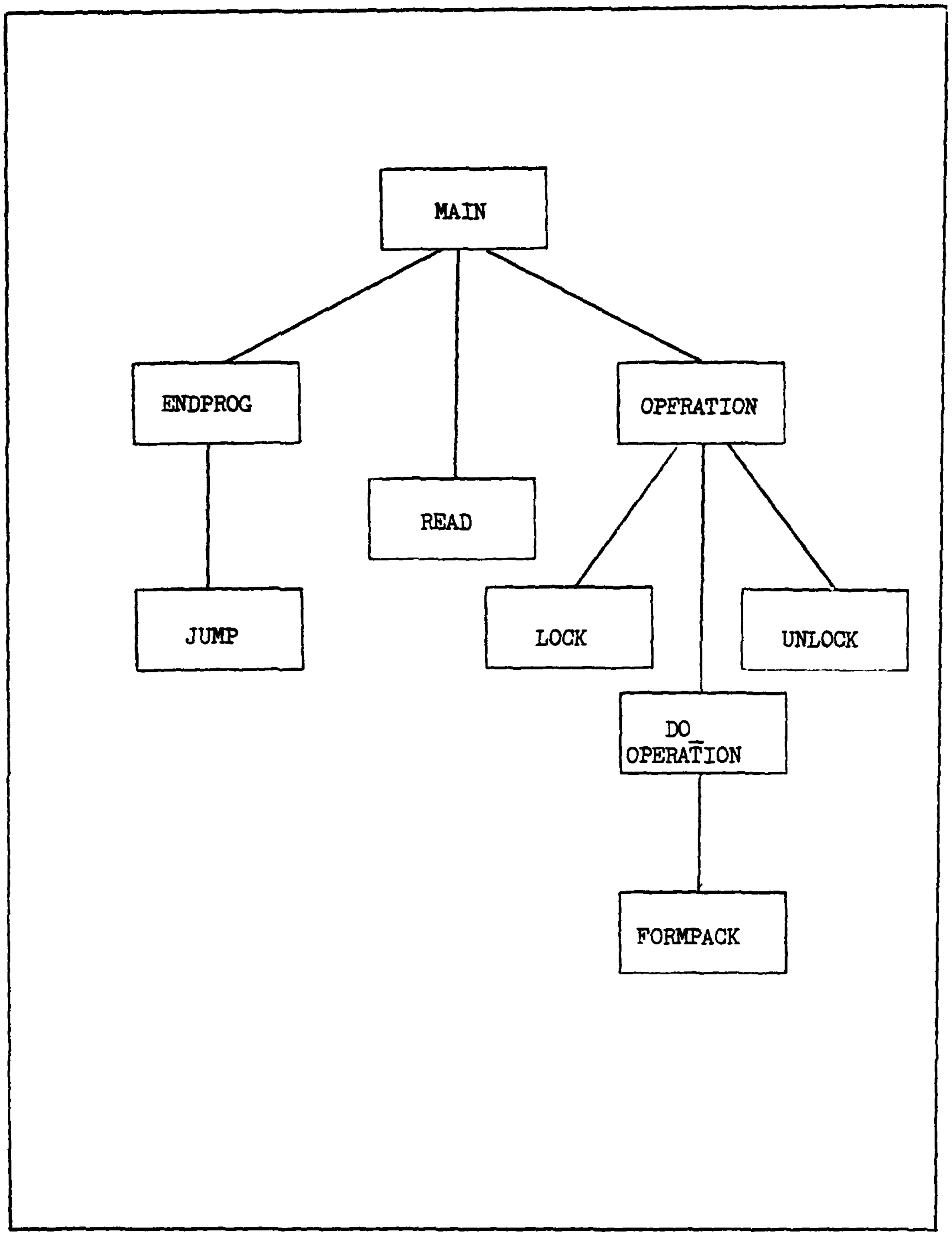


Fig. 26 Slave Processor Structure Chart

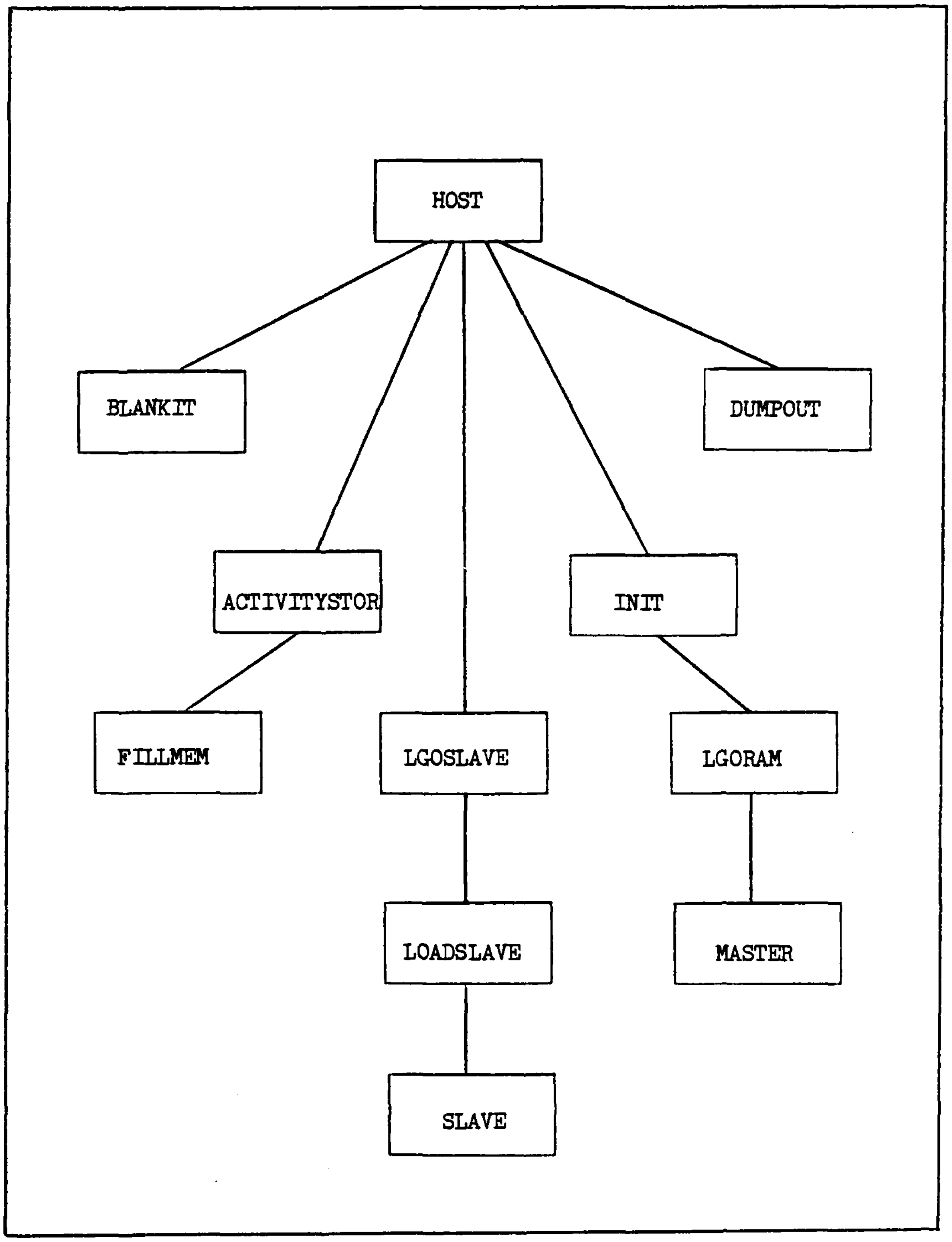


Fig. 27 Host Structure Chart

would be required. Only three were available. However, future investigations could implement a variation of this design by restricting the types of operations performed by each processor. For example, one processor could perform all the arithmetic computations, one could process all comparisions and switching operations, and one could process the I/O.

The second approach was not chosen because of an obvious duplication of effort. Also, the results of Boesch's work (Ref 6) indicated that his design may not have implemented a practical data flow processor. This approach was not without merit. For example, since any node could be processed by any processor, the software had to be identical. This meant that only one set of processor software had to be developed and tested. On the surface, it appeared that this aspect would make the pooled processor approach most attractive. Much time was spent trying to understand the software of the previous investigation. However, due to the absence of documentation, it was determined that this approach should not be implemented!

Summary

This chapter has presented the design for the data flow language used to represent a data flow program to the multiprocessor. Also, the design for the master and slave processors' software has been presented. The next chapter describes the implementation.

VI. Implementation

Introduction

The software modules of the data flow multiprocessor, designed in the previous chapter, were implemented. In addition, the hardware was modified to allow multiprocessing. The unique factors that impacted this implementation are discussed in this chapter as are the basic principles used in implementing the modules. Also, the testing procedures that were employed to verify the modules are summarized. Finally, the testing results are stated.

Hardware Implementation

Before implementing any software, some modifications had to be made to the two SBC 80/20 microprocessors to allow them to multiprocess. Also, two types of interference between multiple processors had to be addressed. The types of interference are: mutual exclusion of the bus, and mutual exclusion of memory.

Bus exclusion means that when one processor needs the system bus to perform input, output, or memory operations, the other processors will be prevented from interfering. The Intel computers have two means of bus arbitration: parallel and serial. Since the serial bus arbitration method could be implemented simply through the use of wire wrap jumpering, it was implemented. In the serial method all masters on the bus are arranged in order of their priority. When a processor wants access to the bus, it checks the bus busy line (BUSY). If the bus is not busy then it raises its bus priority (BPRO) line telling all lower priority masters that the bus is needed, then checks the next higher master on the bus. If the next higher master has not requested the bus before the next falling clock edge, then the processor has access to the bus. It then raises the BUSY line telling all masters that the bus is in use. This serial (daisy-chain) method of bus arbitration is shown in Figure 28 (Ref 2). This method is implemented by grounding the BPRN (Bus Priority In) of the first master, and chaining the BPRO to BPRN signals of all subsequent masters. This is easily done by jumpering the proper posts on the back of the SBC card cage.

Unfortunately, this method results in the first processor on the bus having a higher priority than subsequent processors. For the design of the multiprocessor, this means that the master operations have higher priority over all slave operations. However, since the master processor fetches enabled instructions and updates result packets, it should have higher priority. If the slave processors had higher priority than the master, then no instruction would ever be enabled. The function of the slave processor software is to check for enabled nodes (operation packets), and perform the required operation. With higher priority slaves, the multiprocessor would

AD-A115 616   AIR FORCE INST OF TECH WRIGHT-PATTERSON AFB OH SCHOO--ETC F/G 9/2
THE DESIGN AND IMPLEMENTATION OF A DATA FLOW MULTIPROCESSOR.(U)
DEC 81 M W BRAY
UNCLASSIFIED AFIT/GCS/EE/81D-5   NL

2 OF 2
AD A
115616

END
DATE
FILMED
07-82
DTIC

continously check for enabled nodes that could only be enabled by the master. Therefore, the master processor must be allowed to initially identify all enabled nodes, and to subsequently enable other nodes as they become available. The serial arbitration method results in a higher priority master, and thus assures this function.

The other type of processor interference that had to be addressed was memory exclusion. Memory exclusion means that one processor has the ability to lock other processors out of memory for several instructions. This is done by the use of the bus override command. This command causes the bus interface circuit in the SBC 80/20 to gain command of the bus as described previously. When it has command of the bus however, the circuit does not release the bus. Therefore, all other processors are locked out of the bus. This command allows a processor to safely access and update data in common shared memory. When the critical phase is completed, the processor can issue a bus release and allow other processors access to the bus again. This locking command is implemented through software by writing the value 1 to I/O port "D5". The release command is also implemented through software by writing the value 0 to I/O port "D5". These are the functions of the modules called "LOCK" and "UNLOCK" in the structure charts of Chapter V (Figure 25 and 26).

Some hardware modifications were made to the processors

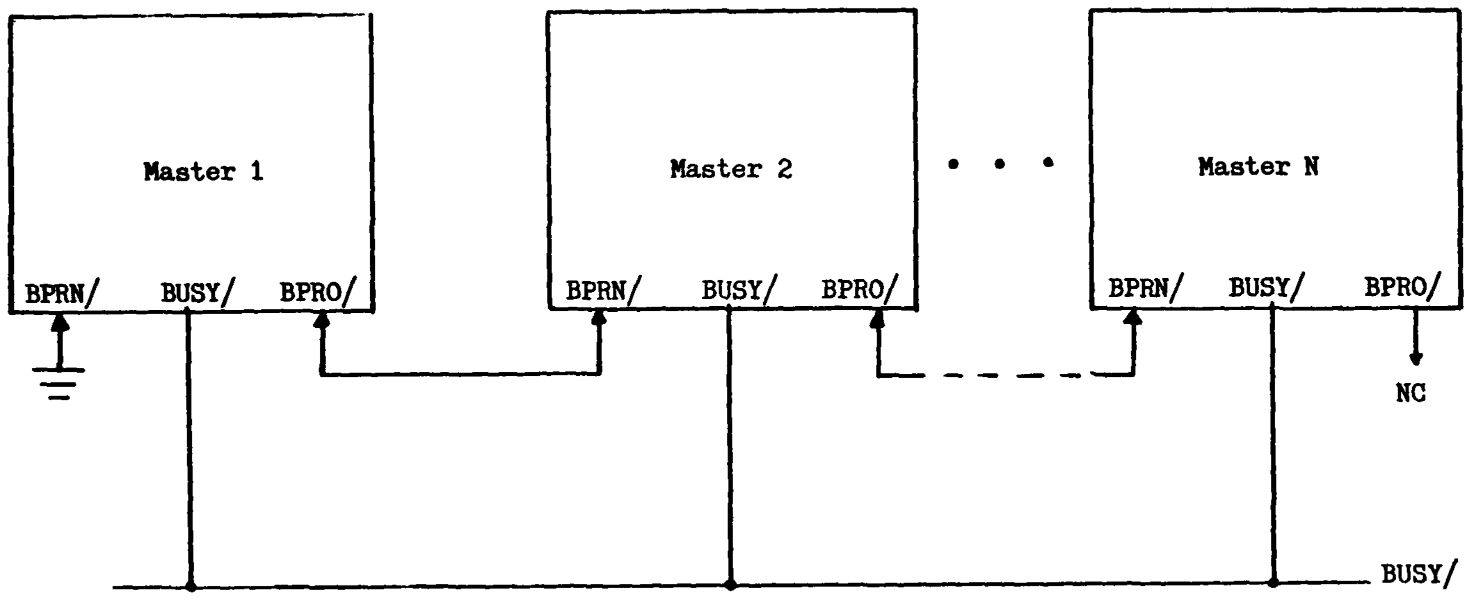


Fig. 28 Serial Bus Arbitration

prior to the implementation of software. There was only one modification to be made to the master processor. The on board random access memory was reconnected so that it appeared in locations F800H to FFFFH. This was accomplished by jumpering wire wrap pin 117 to pin 121. The changes to the slave processor were more substantial. The multibus requires a bus clock and a system clock. Both SBC 80/20s provided this clock therefore, the bus clock on the second processor had to be disabled. The modifications required were: removal of jumper from pins 110 to 111 (Bus clock), and the removal of jumper from pins 111 to 112 (System clock). This was done so that the master processor could provide both of these clocks. The final modification made to the slave was to remove the standard SBC 80/20 ROM monitor. The ROM monitor was replaced with the ROM program of Figure 21 to allow the master to command the slave.

Software Implementation

Before any modules of the data flow multiprocessor were implemented, two software problems had to be resolved. First, due to the large amount of shared memory required, the standard 64k PASCAL system had to be reconfigured. Second, to fully exploit the capabilities of the UCSD PASCAL editor, a smart terminal (H19) had to be configured into the operating system.

The memory map of Table III shows that locations D800H to FFFFH are effectively allocated to functions other than the operating system. Obviously, if the 64k PASCAL operating system were left unchanged the shared memory would be overwritten and unusable. Therefore, a 52k system had to be configured. Since PASCAL requires that CPM be booted first, a 52k CPM system also had to be configured. Both systems are relatively easily configured and both processes are documented in the CPM and UCSD-PASCAL manuals (Ref 25).

Another change made to the PASCAL operating system was the reconfiguration to support an intelligent terminal (H19). When PASCAL was initially configured with an ADM-3 CRT, the editor displayed some undesirable features. For example, when entering programs, a carriage return caused the previously typed line to disappear. This problem did not occur with other laboratory UCSD systems because they were configured with intelligent terminals. Therefore, the H19 direct cursor addressing routine was configured into the PASCAL system (by using the "BINDER" program). The setup code was also changed to reflect the cursor addressing sequences of the H19. Once this terminal was configured into the system, all editor capabilities were available and working properly.

Once the changes to the operating system were made, the software was implemented in a modular top down approach. The structure charts of Chapter V show the modules of the three major software systems: the PASCAL host (driver), the master software, and the slave software. The order of implementation was important due to testing constraints.

First, the PASCAL host software was implemented, followed by the master software, followed by the ROM program for the slaves, and finally the slave processor software. The source code for the multiprocessor software is in Appendix A, along with the user's guide in Appendix B.

Testing

Because of the time limit on this investigation and various implementation problems, the testing was not as rigorous as it could have been. However, various tests were conducted to verify the performance of the multiprocessor software. For example, inputs were chosen so that every segment of code was used. This was done by choosing inputs to cause each branch to be taken. The testing was done incrementally. As a module was coded and debugged, it was then integrated into the set of modules that had already been tested. This new set of modules was then tested by choosing inputs which executed each segment of code.

The PASCAL host software, the master software modules, and the slave software modules were individually tested by verifying their effects on shared memory. The ROM monitor of the slave processor allowed all 64k of RAM memory to be displayed in hex format. By using the slave's ROM monitor to display shared memory, it was easy to verify that the PASCAL host blanked out shared memory, and that the master software properly stored and manipulated the activity store.

However, when the complete multiprocessor software was implemented, and the ROM monitor replaced with the ROM program, this testing technique was no longer available. Testing of the complete multiprocessor software could only be accomplished by the use of a post mortem dump routine which displayed the contents of shared memory.

Initial Results of the Multiprocessor

Various data flow instructions were processed by the multiprocessor throughout the testing. However, the final test to prove the correct functionality was to process an entire data flow program. Two types of data flow programs were tested. One was a simple sequential program made up of arithmetic data flow instructions. The purpose of this type of program was to verify that results from one operation could be correctly passed to other operations. This sequential program produced the correct results and proved that the software was passing results correctly. The second type of program was a looping program. The purpose of this program was to test the performance of the software in a concurrent environment. The data flow looping program of Figure 20 was processed by the data flow multiprocessor. The multiprocessor computed the correct results, thus verifying the functionality of the multiprocessor software. However, execution times could not be measured to validate the execution speedup, if any. This was because there was no programmable clock available. One could be developed with some hardware modifications and a time keeping program stored in ROM. This programmable clock would function by setting the value 0 into the clock as soon as the multiprocessor begins execution. The Intel 8253 (present on both master and slaves) could interrupt the system at a fixed time interval, such as 10 milliseconds, for example. The time keeping program would increment the clock value at every interrupt. At the completion of execution, the clock value would contain the execution speed to the nearest 10 milliseconds.

Summary

After modifying the hardware to enable the SBC's to multiprocess, and after configuring the PASCAL operating system to the correct size, the data flow multiprocessor software was implemented. The software was implemented in a modular top down approach to facilitate ease of implementation and testing. Finally, the data flow multiprocessor was tested by executing a complete data flow program.

VII. Conclusions and Recommendations

The intent of this investigation was to thoroughly research current data flow techniques and to apply them to the design and implementation of a data flow multiprocessor. This report has presented the data flow techniques used to design the multiprocessor, as well as the steps involved in the implementation of the multiprocessor.

Once the literature was researched, a set of structured specifications for the multiprocessor was formulatted. The structured specifications were then translated into a design. The modular top down design approach was used. This forced the interfaces to be defined early. Unfortunately, due to unfamiliarity with microprocessors and multiprocessor communication, much time was spent researching some of these interface design techniques (shared memory, master/slave communication). However, because of the work done in specifying the requirements, the design of the software proved to be fairly straightforward.

Once the software was designed, the hardware had to be modified. The modifications to the hardware were easy to implement. However, in the course of working with the hardware many unfamiliar hardware problems occurred. Debugging these problems resulted in replacing several integrated circuits on each of the two microprocessors. Some software problems also had to be overcome. The operating system had to be configured to allow an unused 12k shared memory block. Also, a change of CRT had to be configured into the operating system. The solution to both hardware and software problems accounted for the most time consuming portions of this effort. Unfortunately, the amount of time required to solve these problems impacted the attainment of goals of this investigation. However, much was learned from the solutions to these problems, and future related research efforts may benefit from them.

When both the hardware and software problems were finally resolved, the multiprocessor software was implemented. This implementation was done in a top down method, and testing was performed on each module implemented. Finally, the data flow multiprocessor was tested by executing a complete data flow program. The multiprocessor produced the correct answers and demonstrated its functionality.

Recommendations

This area has a significant potential for further study. Three areas seem most promising: further development and enhancements of the multiprocessor, language development, and the design of a microprocessor to directly execute data flow programs.

Further Enhancements of the Multiprocessor. To make the multiprocessor more useful and realistic, the following tasks must be accomplished:

1. Implement real number calculations,

2. Implement capability to process procedures,

3. Implement better input/output capability,

4. Implement a programmable clock to test execution speeds.

The multiprocessor design in this investigation allows only integer calculations. This is not practical for scientific computations. Therefore, the multiprocessor should be modified to include real computations. Large data flow programs may be broken down into subroutines or procedures. The multiprocessor has no capability to process procedures. Therefore, a desirable modification would be the ability to process procedures. The multiprocessor only has the capability to read or write to the CRT. The data flow program is loaded into shared memory via the PASCAL host sottware. It would be desirable for the multiprocessor to be able to read from and write to disk files so that the dataflow software could load its own programs, without the aid of PASCAL. Finally, if a programmable clock was implemented, execution speeds could be measured. This would allow the verification of execution speedup, and would aid in making design modifications more efficient (faster).

Language Development. The development of a data flow program is limited by the programmer's ability to express the program in data flow notation. The language defined in this effort is not easy to use. Improvements must be made to the language to allow for better human understanding.

One related method of accomplishing this would be to design and implement a data flow translator. This translator would allow expression of a program in some structured HOL and translate it into data flow notation. Work on a translator definitely merits some investigation.

Microprocessor Design. A chip set could be designed which would directly execute data flow notation. These chips would comprise a microprocessor which would be the most efficient method of processing data flow notation.

Bibliography

1. Ackerman, William B. and Dennis, Jack B. "VAL-A Value Oriented Algorithmic Language: Preliminary Reference Manual," Massachusetts: Laboratory for Computer Science, Massachusetts Institute of Technology, June 1979.

2. Adams, George and Rolander, Thomas "Design Motivations for Multiple Processor Microcomputer Systems," Computer Design, March 1978.

3. Arvind and Gostelow, K. P. "The Preliminary ID Report: An Asynchronous Programming Language and Computing Machine," Department of Computer Science, University of California, Irvine, May 1978.

4. Arvind and Vinod Kathail "A Multiple Processor Data Flow Machine That Supports Generalized Proceedures," 8th Annual Symposium on Computer Architecture, May 1981.

5. Arvind, Professor of Computer Science. Telephone Conversation. Massachusetts Institute of Technology, June 1981.

6. Boesch, Brian P. "Data Flow Techniques in Single-User Multiprocessor Systems," MS Thesis, Air Force Institute of Technology, Dec 1979.

7. Burkowski, Forbes J. "A Multi-User Data Flow Architecture," 8th Annual Symposium on Computer Architecture, May 1981.

8. Denning, Peter J. "Operating System Principles for Data Flow Networks," Computer, July 1978.

9. Dennis, Jack B., Professor of Computer Science. Telephone Conversation. Massachusetts Institute of Technology, June 1981.

10. Dennis, Jack B. "A Computer Architecture for Highly Parallel Signal Processing," Proceedings of 1974 ACM Conference, Nov 1974.

11. Dennis, Jack B. "First Version of a Data Flow Procedure Language," Massachusetts Institute of Technology, May 1975.

12. Dennis, Jack B. "Data Flow Supercomputers," Computer, Nov 1980.

13. Dennis, Jack B. and Misunas, David P. "A Highly Parallel Processor Based on the Data Flow Concept," Laboratory for Computer Science, Massachusetts Institute of

Technology.

14. Dennis, Jack B. and Misunas, David P. "A Preliminary Architecture for a Basic Data Flow Processor," Laboratory for Computer Science, Massachusetts Institute of Technology.

15. Gostelow, Kim and Thomas, Robert "Perfomance of a Simulated Data Flow Computer," _IEEE Transactions on Computers_, Vol. C-29, No. 10, Oct 1980.

16. Hayes, John P. _Computer Architecture and Organization_, McGraw-Hill Book Company, New York, 1974.

17. Leventhal, Lance A. _Z80 Assembly Language Programming_, Osborne & Associates Inc, Berkley, California, 1979.

18. Madnick, Stuart E. and Donovan, John J. _Operating Systems_, McGraw-Hill Book Company, New York, 1974.

19. Miranker, Glen S. "Implementation of Procedures on a Class of Data Flow Processors," _Proceedings of the 1977 International Conference on Parallel Processing_, Aug 1977.

20. Misunas, David P. "A Report on the Workshop on Data Flow Computer and Program Organization," Laboratory for Computer Science, Massachusetts Institute of Technology.

21. Patil, S. and Dennis J. "The Description and Realization of Digital Systems," Massachusetts Institute of Technology.

22. Peterson, James L. "Petri Nets" _Computing Surveys_, Vol. 9, No. 3, Sep 1977.

23. Reghbati, H. and Hamacker, V. "Hardware Support for Concurrent Programming in Loosely Coupled Multiprocessors," _5th Annual Symposium on Computer Architecture_, IEEE, Apr 1978.

24. Rumbaugh, James E. "A Data Flow Multiprocessor," _IEEE Transactions on Computers_, Vol. C-26, No. 2, Feb 1977.

25. _UCSD (Mini-Micro Computer) Pascal Version II.0_, Institute for Information Systems, USCD Mailcode C-021, La Jolla, Ca. 92093, March 1979.

26. Weinberg, Victor _Structured Analysis_, Yourdon Press, New York, 1977.

27. Weng, K. "Stream Oriented Computation in Recursive Dataflow Schemas," Massachusetts: Project MAC, TM-68, Massachusetts Institute of Technology, Oct 1975.

28. Wirth, Niklaus and Jensen, Kathleen *Pascal Users Manual and Report*, Springer-Verlang, New York 1974.

Appendix A

Source Code for the Data Flow Multiprocessor

This appendix contains the PASCAL and assembly language modules that comprise the data flow multiprocessor. Included in this appendix are the HOST software modules, the Master processor modules, the Slave processor modules, and the stand-alone program CONVERT.

```
PROGRAM DATAFLOW;  (* P-MACHINE HOST PROGRAM *)
 TYPE
   FF = RECORD
        NODENUM : INTEGER;
        OPERATOR : CHAR;
        NUMINPUTS,NUMCONSTANTS : INTEGER;
        CONSVALUE,NACKSIG : INTEGER;
        OUTPTS : ARRAY [1..2] OF
                RECORD
                OUTNODENUM : INTEGER;
                INPTNODENUM : INTEGER;
                END;
        ACKNODE1,ACKNODE2 : INTEGER;
        END;

      VAR
        MEM : ARRAY [1..128] OF FF ;
        PROG : PACKED ARRAY [0..2048] OF CHAR ;
        F : FILE OF FF;
        S : STRING;
        C,OP : CHAR;
        I,NUMNODES,J  : INTEGER;
        N1,N2,N3,N4,N5,N6,N7,N8,N9,N10,N11 : INTEGER;


        PROCEDURE BLANKIT; EXTERNAL;   (* BLANKS OUT MEMORY FROM
                                          D800 - F000H               *)


        PROCEDURE DUMPOUT; EXTERNAL; (* DUMPS THE OUTPUT BUFFER *)


        PROCEDURE FILLMEM; EXTERNAL;   (* FILLS THE ACTIVITY STORE
                                          WITH THE DATA FLOW
                                          INSTRUCTIONS.            *)

        PROCEDURE LGORAM; EXTERNAL; (* THIS PROCEDURE LOADS AND THEN
                                        EXECUTES THE MASTER PROCESSOR
                                        SOFTWARE.  *)

        PROCEDURE LOADSLAVE; EXTERNAL; (* THIS PROCEDURE LOADS AND THEN
                                          EXECUTES THE SLAVE SOFTWARE *)

     PROCEDURE LISTFLOW;     (*  THIS PROCEDURE IS CALLED ONLY WHEN
                             WE WANT A LISTING OF ALL THE INPUT DAT
                             FLOW NODES.  THIS IS USEFUL TO MAKE SURE
                             THAT WE HAVE CORRECTLY INPUT THE REQUIRED
```

```
                    DATA FLOW NOTATION.  *)
  VAR
   LIST : INTERACTIVE;

  BEGIN
   RESET(LIST, 'CONSOLE:');
   WRITELN(LIST, 'OUTPUTING DATA FLOWS----');
   FOR I := 1 TO NUMNODES DO
           BEGIN
           WRITE(LIST,' ',MEM[I].OPERATOR,' ',MEM[I].NUMINPUTS,' ',
                   MEM[I].NUMCONSTANTS,' ',
                   MEM[I].CONSVALUE,' ',MEM[I].NACKSIG,' ');
           FOR J := 1 TO 2 DO
                   WRITE(LIST,MEM[I].OUTPTS[J].OUTNODENUM,' ',
                           MEM[I].OUTPTS[J].INPTNODENUM,' ');
           WRITE(LIST,MEM[I].ACKNODE1,' ',MEM[I].ACKNODE2);
           WRITELN(LIST);
           END;
   END;  (* LISTFLOW *)

PROCEDURE ACTIVITYSTORE;
           (*  THIS PROCEDURE PLACES THE DATA FLOW INSTRUCTIONS
               JUST READ INTO THE ACTIVITY STORE.  PROCEDURE
               FILLMEM IS AN ASSEMBLY ROUTINE WHICH ACTUALLY
               STORES THE INSTRUCTIONS.                       *)

   BEGIN (* ACTIVITYSTORE *)
     FOR I := 1 TO NUMNODES DO
       BEGIN (* FOR I *)
           OP := MEM[I].OPERATOR;
           N1 := I - 1; (* FIRST NODE MUST BE 0! *)
           N2 := MEM[I].NUMINPUTS;
           N3 := MEM[I].NUMCONSTANTS;
           N4 := MEM[I].CONSVALUE;
           N5 := MEM[I].NACKSIG;
           N6 := MEM[I].OUTPTS[1].OUTNODENUM;
           N6 := N6 * 16;     (* AIDS ADDRESS CALCULATION IN FILLMEM
                                 IF THE VALUE IS MULTIPLIEDBY 16*)
           N7 := MEM[I].OUTPTS[1].INPTNODENUM;
           N8 := MEM[I].OUTPTS[2].OUTNODENUM;
           N8 := N8 * 16;  (* SAME AS FOR N6 *)
           N9 := MEM[I].OUTPTS[2].INPTNODENUM;
           N10 := MEM[I].ACKNODE1;
           N11 := MEM[I].ACKNODE2;
           J := (I-1)*16;
           FILLMEM;
       END (* FOR I *);
     END; (* ACTIVITYSTORE *)

   PROCEDURE INIT;  (* READS THE MASTER SOFTWARE INTO THE "PROG"
                                   STORAGE ARRAY.  *)
   VAR
           A : FILE;
```

```
            B : INTEGER;
            LIST : INTERACTIVE;
            BEGIN
             RESET (A,'#5:MASTER.CODE');
             RESET(LIST,'CONSOLE:');
             WRITELN (LIST,' READING MASTER SOFTWARE...');
             B :=  BLOCKREAD (A,PROG,3,1);
             WRITELN (LIST,' MASTER SOFTWARE READ.. NOW LOAD IT');
             LGORAM;
             WRITELN (LIST,' MASTER SOFTWARE EXECUTING!!!!');
            END;

    PROCEDURE LGOSLAVE;   (* LOADS AND EXECUTES THE SLAVE *)
    VAR
            A : FILE;
            B : INTEGER;
            LIST : INTERACTIVE;
     BEGIN
            RESET (A,'#5:SLAVE.CODE');
            RESET (LIST,'CONSOLE:');
            WRITELN(LIST,' READING SLAVE SOFTWARE....');
            B := BLOCKREAD (A,PROG,3,1);
            WRITELN (LIST,' SLAVE SOFTWARE READ.. NOW EXECUTE IT');
            LOADSLAVE;
            WRITELN(LIST,'SLAVE SOFTWARE EXECUTING!');
     END;


  BEGIN (* MAIN *)
    WRITE ('ENTER FILE NAME OF DATA FLOW NOTATION ');
    READ (S);
    RESET(F,S);
    NUMNODES := 0;
    WHILE NOT EOF(F) DO       (* INPUT NODES *)
            BEGIN
            NUMNODES := NUMNODES + 1;
            MEM[NUMNODES] := F^  ;
            GET(F);
            END;
            WRITELN(NUMNODES);

    WRITELN(' WRITE INPUT LISTING? [Y/N] ');
    READ(KEYBOARD,C);
    IF C = 'Y' THEN
            LISTFLOW;   (* CALL THE DUMP ROUTINE *)
    BLANKIT;
    ACTIVITYSTORE;
    WRITELN( ' DO YOU WANT THE SLAVE TO BE LOADED? [Y/N]');
    READ(KEYBOARD,C);
    IF C = 'Y' THEN
            LGOSLAVE;
    INIT;
    WRITELN( 'DO YOU WANT THE OUTPUT BUFFER DUMPED? [Y/N]');
```

```
        READ(KEYBOARD,C);
        IF C = 'Y' THEN
                DUMPOUT;
    END.      (* MAIN *)
```

```
        ;*********************************************************
        ;   PROCEDURE BLANKIT
        ;   CALLED BY :    HOST PROGRAM "DATAFLOW"
        ;   CALLS     :    NOTHING
        ;**********************************************************
        .PROC   BLANKIT
                        ; THIS ROUTINE BLANKS OUT MEMORY LOCATIONS
                        ; D800 - F000 HEX.  THESE LOCATIONS MUST BE
                        ; BLANK PRIOR TO PLACING ANYTHING IN THEM.
                        ; THIS IS WHERE THE ACTIVITY STORE WILL
                        ; RESIDE.


BLOK    .EQU    0D800H          ; BEGINNING OF MEMORY TO BLANK
EBLOK   .EQU    0F000H          ; HIGH BOUNDARY OF MEMORY TO GET
                                ; BLANKED
BLANK   .EQU    20H             ; ASCII BLANK


        LD      HL,BLOK         ; PLACE BEGINNING MEM IN HL REG

LOOP1:  LD      (HL),BLANK      ;PLACE BLANK IN THE LOCATION
        INC     HL              ;INCREMENT TO NEXT LOCATION
        LD      A,H             ;LOAD H INTO A FOR COMPARISON
        CP      0F0H            ;CHECK IF WE ARE AT HIGH BOUNDARY
        JP      NZ,LOOP1        ;BRANCH IF NOT
        LD      HL,EBLOK        ;LOAD HIGH LOCATION
        LD      (HL),BLANK      ;PLACE BLANK IN HIGH LOCATION
        RET             ; NOW WE ARE DONE
        .END
        ;***********************************************
        ;  PROCEDURE FILLMEM
        ;  CALLED BY:  HOST PROGRAM "DATAFLOW"
        ;  CALLS    :  NOTHING
        ;***********************************************
        .PROC   FILLMEM         ; THIS PROCEDURE FILLS THE ACTIVITY
                                ; STORE WITH THE DATA FLOW INSTRUCTIONS
        .PUBLIC J,N1,N2,N3,N4,N5,N6,N7
        .PUBLIC N8,N9,N10,N11,OP

LOKLOC  .EQU    0D7F0H
MEMBLK  .EQU    0E000H          ;START OF ACTIVITY STORE

        LD      HL,LOKLOC
        LD      A,0
        LD      (HL),A   ;INITIALIZE LOCK LOCATION WITH 0
        LD      HL,MEMBLK       ;LOAD BASE ADDRESS OF ACTIVITY STORE
        LD      BC,J             ;LOAD DATA FLOW NODE# ADDRESS
        LD      A,(BC)          ;LOAD DATA FLOW NODE# VALUE
        LD      B,00H           ;ZERO OUT HIGH ORDER REGISTER
        LD      C,A
        ADD     HL,BC   ;FIND WHERE THIS NODE IS TO BE STORED IN THE
        LD      BC,N5   ;ACTIVITY STORE
        LD      A,(BC)
        LD      (HL),A          ;LOAD # ACK. SIG. RECEIVED
        INC     HL
        LD      BC,N1
        LD      A,(BC)
        LD      (HL),A          ;LOAD NODE#
        INC     HL
        LD      BC,OP
        LD      A,(BC)
        LD      (HL),A          ;LOAD OPERATOR
        INC     HL
        LD      BC,N2
        LD      A,(BC)
        LD      (HL),A          ;LOAD # OF OPERANDS
        INC     HL
        LD      BC,N3
        LD      A,(BC)
        LD      (HL),A          ;LOAD # OF CONSTANTS

                                ;NOW LOAD DOUBLE BYTES
                                ;WITH THE APPROPRIATE VALUES
        CP      1
        JP      NZ,ZEROCON
        INC     HL
        LD      BC,N4
        LD      A,(BC)
        JP      LOADZERO

ZEROCON
        INC     HL
        LD      BC,N4           ;LOAD OPER#1 ADDRESS
        LD      A,(BC)          ;LOAD OPER#1 VALUE, IF ANY
        CP      0               ;SEE IF THERE ARE ANY VALUES
        JP      Z,NOVALS        ;IF NOT, JUMP
LOADZERO
        LD      (HL),A          ;IF SO, LOAD LOW ORDER BYTE
        INC     HL
        INC     BC              ;GET HIGH ORDER VALUE
        LD      A,(BC)          ;
        LD      (HL),A          ;LOAD HIGH ORDER BYTE
                                ;OPER#1 IS NOW LOADED
        INC     HL
        LD      (HL),0FFH       ;NOW LOAD FFFFH INTO OPER#2
        INC     HL
        LD      (HL),0FFH
        INC     HL
        JP      VALUES
NOVALS                          ;JUMP HERE IF NO OPER#1 VALUE
                                ;LOAD FFFFH INTO OPER#1 AND 2
        LD      (HL),0FFH
        INC     HL
        LD      (HL),0FFH
        INC     HL
        LD      (HL),0FFH       ;LOAD FFFFH INTO OPER#2
        INC     HL
        LD      (HL),0FFH
        INC     HL
VALUES                          ;JUMP HERE IF THERE WAS OPER#1 VALUE
                                ;NOW COMPUTE DEST#1 AND DEST#2 ADDRESSES
                                ;FIRST CHECK N7 TO SEE IF THERE IS A
                                ;DEST#1 ADDRESS
        LD      BC,N7
        LD      A,(BC)          ;LOAD A WITH VALUE OF N7
        CP      0               ;IF N7=0 THEN NO ADDRESS NEED BE COMPUTED
        JP      Z,NOADD1        ;JUMP IF NO COMPUTATION TO BE DONE
        CP      1       ;IF N7=1, THEN INPUT #=1 THEREFORE ADD 5 TO N6
        JP      NZ,DEST1_2      ;OTHERWISE JUMP AND ADD 7 TO N6
        LD      BC,N6           ;LOAD WHERE N6 IS
        EX      DE,HL           ;SAVE HL INTO DE FOR LATER USE
        LD      A,(BC)          ;LOAD LOW ORDER N6 VALUE
        LD      L,A
        INC     BC
        LD      A,(BC)          ;LOAD HIGH ORDER N6 VALUE
        LD      H,A
        LD      BC,MEMBLK
        ADD     HL,BC           ;COMPUTE START ADDRESS + OFFSET
        LD      BC,0005
        ADD     HL,BC           ;ADD 5 TO GET CORRECT ADDRESS
        EX      DE,HL           ;RESTORE HL TO ORIGINAL VALUE
        LD      A,E             ;LOAD LOW ORDER BYTE
        LD      (HL),A          ;
        INC     HL
        LD      A,D             ;LOAD HIGH ORDER BYTE
        LD      (HL),A
        INC     HL
        JP      ADDR2           ;NOW COMPUTE DEST#2 ADDRESS

DEST1_2                         ;ADD 7 TO N6 TO COMPUTE CORRECT ADDR
        LD      BC,N6           ;LOAD WHERE N6 IS
        EX      DE,HL           ;SAVE HL
        LD      A,(BC)          ;LOAD LOW ORDER N6 VALUE
        LD      L,A
        INC     BC
        LD      A,(BC)          ;LOAD HIGH ORDER N6 VALUE
        LD      H,A
        LD      BC,MEMBLK
        ADD     HL,BC           ;COMPUTE START ADDRESS + OFFSET
        LD      BC,0007
        ADD     HL,BC           ;ADD 7 TO COMPUTE CORRECT ADDRESS
        EX      DE,HL           ;RESTORE HL TO ORIGINAL VALUE
        LD      A,E
        LD      (HL),A          ;LOAD LOW ORDER BYTE
        LD      A,D
        INC     HL
        LD      (HL),A          ;LOAD HIGH ORDER BYTE
        INC     HL
        JP      ADDR2           ;NOW COMPUTE DEST#2 ADDRESS

NOADD1                          ;JUMP TO HERE IF NO ADDRESS IS TO
                                ;BE COMPUTED.  ZERO OUT THE 2 BYTE
                                ;DEST#1 ADDRESS.
        LD      (HL),00
        INC     HL
        LD      (HL),00
        INC     HL

ADDR2                           ;NOW COMPUTE DEST#2 ADDRESS, IF ANY
                                ;FIRST CHECK VALUE OF N9
                                ;N9=0 NO ADDRESS NEED BE COMPUTED
        LD      BC,N9
        LD      A,(BC)          ;LOAD VALUE OF N9
        CP      0               ;IF N9=0 NO ADDRESS
        JP      Z,NOADD2        ;JUMP IF NO ADDRESS, OTHERWISE
        CP      1               ;IF N9=1 ADD 5 TO N8
        JP      NZ,DEST2_2
        LD      BC,N8           ;LOAD N8 ADDRESS
        EX      DE,HL           ;SAVE HL
        LD      A,(BC)          ;LOW ORDER N8 VALUE
        LD      L,A
        INC     BC
        LD      A,(BC)          ;HIGH ORDER N8 VALUE
        LD      H,A
        LD      BC,MEMBLK       ;LOAD OFFSET
```

```
          ADD     HL,BC              ;COMPUTE START ADDRESS + OFFSET
          LD      BC,0005            ;ADD 5 TO HL
          ADD     HL,BC
          EX      DE,HL              ;RESTORE HL TO ORIGINAL VALUE
          LD      A,E                ;LOAD LOW ORDER VALUE
          LD      (HL),A
          LD      A,D                ;LOAD HIGH ORDER VALUE
          INC     HL
          LD      (HL),A
          INC     HL
          JP      ACKSIG             ;NOW LOAD ACK. SIG. REQUIRED TO FIRE NODE

DEST2_2
          LD      BC,N8              ;THE FOLLOWING IS THE
          EX      DE,HL              ;SAME CODE FOR COMPUTING
          LD      A,(BC)             ;THE ADDRESSES AS WAS DONE
          LD      L,A                ;PREVIOUSLY, EXECT WE ADD
          INC     BC                 ;7 INSTEAD OF 5
          LD      A,(BC)
          LD      H,A
          LD      BC,MEMBLK
          ADD     HL,BC
          LD      BC,0007
          ADD     HL,BC
          EX      DE,HL
          LD      A,E
          LD      (HL),A
          INC     HL
          LD      A,D
          LD      (HL),A
          INC     HL
          JP      ACKSIG             ;NOW LOAD ACKSIG

NOADD2                               ;JUMP TO HERE IF NO DEST#2 ADDR
                                     ;ZERO OUT THE 2 BYTE ADDRESS
          LD      (HL),00
          INC     HL
          LD      (HL),00
          INC     HL

ACKSIG                               ;CONTINUE
          LD      BC,N5
          LD      A,(BC)
          LD      (HL),A            ;LOAD # ACK. SIG. REQ. TO FIRE
          INC     HL
          LD      BC,N10
          LD      A,(BC)
          LD      (HL),A           ;SEND ACK. SIG. TO NODE#
          INC     HL
          LD      BC,N11
          LD      A,(BC)
          LD      (HL),A           ;SEND ACK. SIG. TO NODE#
                                     ;
                                     ; DUMP THE RECORD WILL GO HERE
```

```
RET
.END
```

```
;*****************************************************************
; LOADSLAVE
;CALLED BY:  HOST PROGRAM "DATAFLOW"
;CALLS     :  NOTHING
;  THIS PROCEDURE LOADS THE SLAVE SOFTWARE INTO HIGH RAM
;  F800 - FFFFH OF THE SLAVE PROCESSOR.  WHEN THE SOFTWARE
;  IS LOADED, THE PROCEDURE JUMPS TO HIGH RAM AND ENABLES
;  THE SLAVE PROCESSOR.  THE SLAVE EXECUTES INDEPENDENTLY
;  OF THE HOST AND MASTER.
;*****************************************************************
        .PROC   LOADSLAVE
        .PUBLIC PROG        ;PROG CONTAINS THE SLAVE S/W
RAM     .EQU    0F800H
KEY     .EQU    0F7FDH
LOC     .EQU    0F7FEH

        LD      HL,MOVE
        LD      (LOC),HL
        LD      A,1
        LD      (KEY),A
        LD      A,0
A1      ADD     HL,HL
        DEC     A
        JP      NZ,A1
        RET
MOVE
        LD      HL,RAM
        LD      BC,PROG
        LD      DE,7FFH
LOOP    LD      A,(BC)
        LD      (HL),A
        INC     HL
        INC     BC
        DEC     DE
        LD      A,D
        OR      E
        JP      NZ,LOOP
        JP      RAM
        .END
        ;***************************************************
        ; PROCEDURE LGORAM
        ; CALLED BY:  HOST PROGRAM "DATAFLOW"
        ; CALLS    :  NOTHING
        ; THIS PROCEDURE LOADS THE MASTER S/W INTO THE MASTER
        ; PROCESSORS HIGH RAM AND THEN ENABLES THE MASTER.
        ; THE MASTER EXECUTES UNTIL NO MORE ENABLED DATA FLOW
        ; INSTRUCTIONS ARE PRESENT.
        ;**************************************************************
        .PROC   LGORAM
        .PUBLIC PROG   ;PROG CONTAINS THE MASTER S/W
STORAGE .EQU    0F800H
ENDFLAG .EQU    0F000H
CHAR    .EQU    0CCH
        LD      HL,STORAGE
        LD      BC,PROG
        LD      DE,7FFH
LOOP:
        LD      A,(BC)
        LD      (HL),A
        INC     HL
        INC     BC
        DEC     DE
        LD      A,D
        OR      E
        JP      NZ,LOOP
        LD      HL,ENDFLAG
        LD      (HL),CHAR
        CALL    STORAGE
        RET
        .END
```

```
;****************************************
;  DUMPOUT PROCEDURE
;  CALLED BY:  HOST PROGRAM  "DATAFLOW"
;  CALLS     :  OUTCHAR, CONVERT, VALUETEST
;  THIS PROCEDURE DUMPS THE CONTENTS OF THE OUTPUT BUFFER
;  TO THE CRT IN HEX FORMAT!!  ALL VALUES DISPLAYED ARE IN
;  HEX NOT DECIMAL.
;****************************************
        .PROC   DUMPOUT
COUT    .EQU    0ECH     ;OUTPUT PORT
CONST   .EQU    0EDH     ;CONSOLE STATUS INPUT PORT
TXRDY   .EQU    1        ;TRANSMITTER STATUS MASK
OUTBUF  .EQU    0D800H
CHAR    .BYTE   0
LOW     .BYTE   0
BLANK   .EQU    20H
LMASK   .EQU    0FH
HMASK   .EQU    0F0H

        LD      B,TXRDY
        LD      HL,OUTBUF
LOOP    LD      A,(HL)  ;LOAD A WITH CHAR IN OUTBUF
        CP      20H  ;COMPARE IF IT IS BLANK
        RET     Z        ;IF SO, OUTBUF EMPTY RETURN
        CALL    CONVERT   ;OTHERWISE WRITE NEXT GROUP OF 4 BYTES
        INC     HL
        LD      A,BLANK   ;PUT SPACE BETWEEN VALUES
        LD      (CHAR),A
        CALL    OUTCHAR
        CALL    CONVERT
        INC     HL
        LD      A,(HL)
        LD      (CHAR),A
        CALL    OUTCHAR  ;THIRD BYTE IS A CR
        INC     HL
        LD      A,(HL)
        LD      (CHAR),A
        CALL    OUTCHAR  ;FOURTH BYTE IS A LF
        INC     HL
        JP      LOOP
OUTCHAR
        IN      A,(CONST)
        AND     B
        JP      Z,OUTCHAR  ;UNTIL STATUS READY, KEEP TESTING
        LD      A,(CHAR)  ;LOAD A WITH CHAR
        OUT     (COUT),A   ;OUTPUT CHAR
        RET
CONVERT
```

```
        LD      A,(HL)
        LD      (LOW),A
        AND     HMASK
        CP      0
        JP      Z,LOWTEST  ;NO HIGH ORDER VALUE
        CALL    GETHIGH   ;GET HIGH ORDER VALUE
        CALL    VALUETEST   ;OTHERWISE TEST IF VALUE IS LETTER OR DIGIT
LOWTEST
        LD      A,(LOW)
        AND     LMASK
        CALL    VALUETEST
        RET
VALUETEST
        CP      0AH   ;TEST FOR LETTERS
        JP      M,DIGITS   ;IF NEG, THEN VALUE IS A DIGIT
        ADD     A,37H    ;OFFSET TO CONVERT LETTER TO HEX LETTER
        LD      (CHAR),A
        CALL    OUTCHAR
        RET
DIGITS  ADD     A,30H   ;OFFSET TO CONVERT DIGIT TO HEX DIGIT
        LD      (CHAR),A
        CALL    OUTCHAR
        RET
GETHIGH   ;GET HIGH VALUE. DIVISION BY 16 IS SAME AS RIGHT SHIFT
          ;FOUR TIMES. SO HIGH VALUE IS PUT IN RIGHTMOST 4 BITS
        LD      C,0    ;QUOTIENT
        LD      E,10H   ;DIVISOR IS 16
        LD      D,A   ;DIVIDEND IS HIGH BYTE
L1
        LD      A,E
        CPL
        ADD     A,1   ;TWO'S COMP REP. OF 16
        ADD     A,D   ;ACTUALLY SUBTRACTION
        LD      D,A
        INC     C
        CP      E
        JP      P,L1  ;REPEAT SUBTRACTION UNTIL REMAINDER
        JP      Z,L1  ;LESS THAN 16
        LD      A,C
        RET
        .END
```

```
;************************************************
;               MASTER PROCESSOR
;                  SOFTWARE
; ENABLED BY: LGORAM
; CALLS      : ENABLE,RESULT
;************************************************
        .PROC   MASTER
LOKLOC  .EQU    0D7F0H
ENABCT  .EQU    0D7FFH
OUTBUF  .EQU    0D800H          ;LOCATION OF OUTPUT BUFFER
OPER_P  .EQU    0E800H          ;START OF OPERATION PACKET
RES_P   .EQU    0E80AH          ;START OF RESULT PACKET
MEMBLK  .EQU    0E000H          ;START OF ACTIVITY  STORE
CNOUTST .EQU    0EFH
CNOUT   .EQU    0EEH
TXRDY   .EQU    1
ENDFLAG .EQU    0F000H          ;END_FLAG LOCATION
        .ORG    0F800H
MEMLOC  .WORD   0000H
TRAN1   .BYTE   0
TRAN2   .BYTE   0
DUMM    .BYTE   0
        LD      A,0
        LD      (ENABCT),A      ;INITIALIZE ENABLE NODE COUNT WITH 0
        LD      HL,ENDFLAG
        LD      (HL),A
ENABLECHK                        ;START OF THE ROUTINE
        LD      HL,MEMBLK       ;LOAD BASE ADDRESS OF ACTIVITY STORE
AGAIN   LD      (MEMLOC),HL     ;SAVE CURRENT HL LOCATION
        LD      A,(HL)          ;LOAD #ACK. SIG. RECEIVED
        CP      20H
        JP      Z,NEXT
        CP      0FFH
        JP      Z,NEXT
        LD      BC,000DH        ;LOAD OFFSET TO GET
        ADD     HL,BC           ;#ACK SIG. REQ. TO FIRE NODE
        CP      (HL)             ;IF THEY ARE NOT=NODE CANT BE ENABLED
        JP      NZ,NEXT
                                ;IF THEY ARE=, TEST TO SEE IF
                                ;OPERAND(S) ARE PRESENT
        LD      HL,(MEMLOC)
        LD      BC,0003
        ADD     HL,BC
        LD      A,(HL)
        CP      2
        JP      Z,TWOINP
        INC     HL
        LD      A,(HL)
        CP      1
        JP      NZ,OP_PRES
        CALL    ENABLE
        JP      NEXT
OP_PRES INC     HL
        INC     HL
        LD      A,(HL)
        CP      0FFH
        CALL    NZ,ENABLE
        JP      NEXT

TWOINP
        LD      HL,(MEMLOC)     ;RESTORE LOCATION
        LD      BC,0006         ;OFFSET TO GET HIGH ORDER BYTE
        ADD     HL,BC           ;OF OPER#1
        LD      A,(HL)          ;LOAD IT IN ACC
        CP      0FFH            ;IF=FFH THEN NO OPERAND PRESRENT
        JP      Z,NEXT          ;OTHERWISE TEST IF OPER#2 PRESENT
        LD      BC,0002
        ADD     HL,BC
        LD      A,(HL)
        CP      0FFH            ;MAKE SAME COMPARISION
        CALL    NZ,ENABLE        ;IF=WE FOUND AN ENABLED NODE
                                ;IF NOT ADD 16 TO MEM LOCATION
                                ;AND TEST NEXT NODE
NEXT
        LD      HL,(MEMLOC)     ;RESTORE
        LD      BC,0010H        ;OFFSET OF 16
        ADD     HL,BC           ;HL NOW CONTAINS ADDR OF NEXT NODE TO
                                ;BE TESTED
        PUSH    HL ;SAVE HL SINCE RESULT MIGHT DESTROY IT
        LD      A,(HL)          ;FIRST TEST IF FIELD 0 IS BLANK
        CP      20H             ;IF SO, WE HAVE REACHED THE LAST NODE
                                ;IN THE ACTIVITY STORE SO READ RESULTS
        CALL    Z,RESULT        ;IF NOT TEST IF WE ARE AT THE END OF
                                ;THE ACTIVITY STORE-LOC E800H
        POP     HL  ;RESTORE HL
        LD      A,(0F000H)
        CP      "#"
        RET     Z
        LD      A,(HL)
        CP      20H
        JP      Z,ENABLECHK
        JP      AGAIN         ;IF NOT, READ NEXT NODE
;*******************
;*******************
ENABLE
        CALL    LOCK
        LD      HL,OPER_P       ;LOAD HL WITH SECOND LOC OF OP PACKET
        INC     HL
LOOP    LD      A,(HL)          ;FIND FIRST BLANK SPACE
        CP      20H
        JP      Z,FILL          ;IF = THEN FILL OPERATION PACKET
        LD      BC,0010H        ;IF NOT FIND NEXT LOCATION
        ADD     HL,BC
        JP      LOOP
FILL
        DEC     HL
        CALL    OPPACK
        CALL    UNLOCK
        RET
        ;************
        ;************
OPPACK
        EX      DE,HL
        LD      HL,(MEMLOC)     ;RESTORE LOCATION
        EX      DE,HL
        EX      DE,HL           ;SAVE HL
        LD      BC,0002         ;OFFSET
        ADD     HL,BC
        EX      DE,HL           ;RESTORE HL
        LD      A,(DE)          ;GET OPCODE
        LD      (HL),A          ;LOAD IN LOC 0 OF OPERATION PACKET
        EX      DE,HL
        LD      BC,0003         ;OFFSET FOR OPER#1
        ADD     HL,BC
        EX      DE,HL
        LD      A,(DE)          ;LOAD LOW ORDER BYTE OF OPER#1
        INC     HL
        LD      (HL),A          ;LOAD IN LOC 1 OF OP PACKET
        INC     HL
        INC     DE
        LD      A,(DE)          ;LOAD HIGH ORDER BYTE OPER#1
        LD      (HL),A          ;IN LOCATION 2 OF OP PACKET
        INC     HL
        INC     DE
        LD      A,(DE)          ;LOAD LOW ORDER BYTE OF OPER#2
        LD      (HL),A          ;IN LOC 3 OF OP PACKET
        INC     HL
        INC     DE
        LD      A,(DE)          ;LOAD HIGH ORDER BYTE OF OPER#2
        LD      (HL),A          ;IN LOC 4 OF OP PACKET
        INC     HL
        INC     DE
        LD      A,(DE)          ;LOAD LOW ORDER BYTE DEST#1 ADDR
        LD      (HL),A          ;IN LOC 5 OF OP PACKET
        INC     HL
        INC     DE
        LD      A,(DE)          ;LOAD HIGH ORDER BYTE DEST#1 ADDR
        LD      (HL),A          ;IN LOC 6 OF OP PACKET
        INC     HL
        INC     DE
        LD      A,(DE)          ;LOAD LOW ORDER BYTE DEST#2 ADDR
        LD      (HL),A          ;IN LOC 7 OF PACKET
        INC     HL
        INC     DE
        LD      A,(DE)          ;LOAD HIGH ORDER BYTE
        LD      (HL),A          ;IN LOC 8 OF PACKET
        INC     HL
        EX      DE,HL
        LD      HL,(MEMLOC)     ;RESTORE BEG. ADDR OF NODE
        EX      DE,HL
        INC     DE
        LD      A,(DE)          ;LOAD NODE#
        LD      (HL),A          ;IN LOC 9 OF OP PACKET
                ;NOW WE SEE IF THE ENABLED NODE WAS A "C" NODE.
                ;IF SO, WRITE FFH TO ALL 16 BYTES
        LD      HL,(MEMLOC)     ;RESTORE HL TO POINT TO NODE
        LD      BC,0002         ;OFFSET TO GET TO
        ADD     HL,BC           ;OPCODE
        LD      A,(HL)          ;LOAD OPCODE
        CP      "O"   ;OUTPUT OPCODE DOES NOT GENERATE A RES PACK
                      ;THEREFORE DONT CALL IENCT
        CALL    NZ,IENCT
        LD      A,(HL)   ;RELOAD OPCODE
        CP      "C"
        JP      Z,REMOVE
        CALL    CLEAN           ;OTHERWISE CLEAN THE NODE UP
        RET
REMOVE
        LD      HL,(MEMLOC)     ;GET TO 0TH LOCATION OF NODE
        LD      A,16            ;LOAD ACC WITH COUNTER
LOOP16  LD      (HL),0FFH
        INC     HL
        DEC     A
        CP      0
        JP      NZ,LOOP16
        RET
CLEAN                   ;CLEAN ENABLED NODES BY REMOVING NON-CONSTANT
                        ;OPERANDS AND CHANGE #ACK. SIG. REC. TO 0
        LD      HL,(MEMLOC)
        LD      (HL),00         ;ZERO OUT FIELD 0
        LD      BC,0004         ;OFFSET TO GET # OF CONSTANTS
        ADD     HL,BC
        LD      A,(HL)
        CP      0               ;IF A=0 THEN REMOVE BOTH OPERANDS
        JP      NZ,ONE          ;ELSE REMOVE ONLY THE SECOND ONE
        INC     HL
        LD      (HL),0FFH       ;REMOVE OPER#1
        INC     HL
        LD      (HL),0FFH
        INC     HL
        LD      (HL),0FFH       ;REMOVE OPER#2
        INC     HL
        LD      (HL),0FFH       ;  "     "
        RET
ONE
        LD      BC,0003         ;LOAD OFFSET TO GET TO OPER#2
        ADD     HL,BC
        LD      (HL),0FFH       ;REMOVE OPER#2
        INC     HL
        LD      (HL),0FFH       ;  "      "
        RET
IENCT                   ;INCREMENT ENABLED_NODE_COUNT
        LD      A,(ENABCT)      ;GET VALUE
        INC     A
        LD      (ENABCT),A      ;STORE IT
        RET             ;RETURN TO PROCESS MORE OF THE ACTIVITY STORE
;******************
;  RESULT PACKET PROCEDURE
;******************
RESULT
        CALL    LOCK
        LD      A,(ENABCT)
        CP      0
        JP    Z,DONE
        LD      HL,RES_P
RECHECK
        LD      (MEMLOC),HL
        LD      BC,0002
        ADD     HL,BC
        LD      A,H
        CP      0F0H
        JP      NZ,LLL
        CALL    UNLOCK
        RET
LLL
        INC     HL
        LD      A,(HL)
        CP      20H
        JP      NZ,PROCESS
        LD      HL,(MEMLOC)
        LD      BC,0010H
        ADD     HL,BC
        JP      RECHECK
PROCESS
        DEC     HL
        DEC     HL
        DEC     HL
        LD      C,(HL)  ;LOAD LOW BYTE OF VALUE
        INC     HL
        LD      B,(HL)  ;LOAD HIGH BYTE OF VALUE
        INC     HL
        LD      E,(HL)  ;LOAD LOW BYTE OF DEST#1 ADDR
        INC     HL
        LD      D,(HL)  ;LOAD HIGH BYTE OF DEST#1 ADDR
        LD      A,D
        CP      0
        JP      NZ,LADDR1  ;LOAD 1ST ADDR
        INC     HL  ;OTHERWISE LOAD 2ND ADDR
        JP      LADDR2
LADDR1  EX      DE,HL
        LD      (HL),C ;PLACE VALUE IN PROPER DEST.
        INC     HL
        LD      (HL),B
        EX      DE,HL
        INC     HL
LADDR2  LD      E,(HL)
        INC     HL
        LD      D,(HL)
        LD      A,D
        CP      0
        JP      Z,LBK
        EX      DE,HL
        LD      (HL),C
        INC     HL
        LD      (HL),B
        EX      DE,HL
LBK     CALL    BLANK
        CALL    ACKSIG
        CALL    DENBCT
        CALL    UNLOCK
        RET
BLANK   ;BLANKS OUT RESULT PACKET LOC E__A TO E__F
        LD      HL,(MEMLOC) ;RESTORE HL TO RES PACK BOUNDARY
        LD      A,20H  ;LOAD ACC WITH BLANK
        LD      (HL),A
        INC     HL
        LD      (HL),A
        INC     HL
        LD      (HL),A
        INC     HL
        LD      (HL),A
        INC     HL
        LD      (HL),A
        INC     HL
        LD      (HL),A
        RET
ACKSIG   ;UPDATES THE ACQUISITION SIGNALS
        LD      HL,(MEMLOC)
        DEC     HL  ;GET NODE#
        LD      D,(HL)  ;SAVE IT IN D
        LD      HL,MEMBLK   ;HL POINTS TO START OF ACTIVITY STORE
        LD      A,D
        CP      0    ;IF NODE#=0,THEN NO COMP. MUST BE DONE TO
                     ;FIND PROPER INSTRUCTION
        JP      Z,ZNODE
        CALL    MULT   ;THIS ROUTINE CALCULATES THE PROPER INST.
ZNODE   LD      BC,000EH  ;OFFSET TO GET 1ST TRANS. TO NODE# VALUE
        ADD     HL,BC
        LD      A,(HL)
        CP      0FFH
        RET     Z
        LD      (TRAN1),A   ;SAVE IT IN TRAN1
        INC     HL
        LD      A,(HL)   ;GET TRAN2
        LD      (TRAN2),A  ;SAVE IT
        LD      A,(TRAN1)
        CP      0  ;IF TRAN1 =0 THEN DONT UPDATE ANY ACK. SIG.
        RET     Z
        LD      HL,MEMBLK
        CALL    MULT
        LD      A,(HL)
        INC     A
        LD      (HL),A
        LD      HL,MEMBLK
        LD      A,(TRAN2)
        CP      0
        RET     Z
        CALL    MULT
        LD      A,(HL)
        INC     A
        LD      (HL),A
        RET
MULT
        LD      C,10H
        LD      B,0
M1      ADD     HL,BC
        DEC     A
        CP      0
        JP      NZ,M1
        RET
DENBCT
        LD      A,(ENABCT)
        DEC     A
        LD      (ENABCT),A
        RET
DONE
        LD      HL,ENDFLAG
        LD      A,"#"
        LD      (HL),A
        CALL    UNLOCK
        RET
LOCK
        LD      A,64
DCA     DEC     A
        CP      0
        JP      NZ,DCA
        LD      HL,LOKLOC
        LD      A,(HL)
        CP      1
        JP      Z,LOCK
        LD      (HL),A
        RET
UNLOCK
        LD      HL,LOKLOC
        LD      A,0
        LD      (HL),A
        RET
        .END
```

```
;***************************************************
;
;                SLAVE PROCESSOR SOFTWARE
;        ENABLED BY : LOADSLAVE
;        CALLS : LOCK, OPERATION, UNLOCK
;***************************************************
        .PROC   SLAVE
ROM     .EQU    0
LOKLOC  .EQU    0D7F0H
OUTBUF  .EQU    0D800H
OPER_P  .EQU    0E800H
RES_P   .EQU    0E80AH
END_FLAG    .EQU        0F000H
BLANK   .EQU    20H

        .ORG    0F800H
OPCODE  .BYTE   0
DUM     .BYTE   0
OPERAND1        .WORD   0000
OPERAND2        .WORD   0000

ENDPROG
        LD      HL,END_FLAG     ;CHECK IF DONE?
        LD      A,(HL)
        CP      "#"
        JP      NZ,READ  ;IF NOT, FIND 1ST OPERATION PACKET
        JP      ROM
READ    LD      HL,OPER_P  ;FIND 1ST NON-BLANK OP PACK
R1      LD      A,(HL)
        CP      20H
        JP      NZ,READOP  ;IF NONBLANK READ OP PACK
        LD      BC,0010H  ;IF BLANK, GET NEXT LOCATION
        ADD     HL,BC
        LD      A,H
        CP      0F0H
        JP      Z,READ
        JP      R1
READOP  LD      A,(HL)
        LD      (OPCODE),A  ;SAVE OPCODE SINCE IT WILL BE OVERWRITTEN
        CALL    LOCK
        LD      (HL),BLANK  ;STORE BLANK IN LOC 1 SO NEXT TIME THRU
                        ;OP PACKET APPEARS TO BE BLANK
        CALL    OPERATION
        CALL    UNLOCK
        JP      ENDPROG
;***********
;   OPERATION PROCEDURE
;***********
```

```
OPERATION
        LD      A,(OPCODE)  ;RETREIVE OPCODE FOR FINDING OPERATION
        CP      "C"
        JP      Z,CONSTANT
        CP      "E"   ;EXPONENTIATION?
        JP      Z,EXPON
        CP      "R"      ;REPLICATE
        JP      Z,REPLIC
        CP      "I"      ;INPUT
        JP      Z,INPUT
        CP      "O"      ;OUTPUT
        JP      Z,OUTPUT
        CP      "S"      ;SWITCH
        JP      Z,SWITCH
        CP      "+"      ;ADDITION
        JP      Z,ADDIT
        CP      "-"      ;SUBTRACTION
        JP      Z,SUBT
        CP      "*"      ;MULTIPLICATION
        JP      Z,MULT
        CP      "/"      ;DIVISION
        JP      Z,DIVIDE
        CP      ">"      ;GREATER THAN
        JP      Z,GREAT
        CP      "<"      ;LESS THAN
        JP      Z,LESS
        CP      "#"      ;NOT EQUAL TO
        JP      Z,NOTEQ
        CP      "="      ;EQUAL TO
        JP      Z,EQUAL
        RET
CONSTANT
        LD      D,H  ;PUT COPY OF HL INTO DE
        LD      E,L
        LD      BC,000AH
        ADD     HL,BC  ;HL NOW POINTS TO RESULT PACKET
        EX      DE,HL
        INC     HL
        LD      A,(HL)  ;GET LOW ORDER BYTE OF OPERAND
        LD      (DE),A  ;LOAD IT IN RESULT PACKET
        INC     HL
        INC     DE
        LD      A,(HL)  ;GET HIGH ORDER BYTE
        LD      (DE),A
        INC     HL
        INC     HL
        INC     HL  ;HL NOW POINTS TO DEST#1 ADDR IN OP PACK
        INC     DE  ;DE NOW POINTS TO DEST#1 ADDR IN RES PACK
        CALL    FORMPACK
        RET
EXPON
        LD      D,H  ;PUT COPY OF HL INTO DE
        LD      E,L
        INC     HL
        LD      A,(HL)  ;LOAD LOW BYTE OF OPER#1
        LD      C,A
        INC     HL
        LD      B,(HL)  ;LOAD HIGH BYTE OF OPER#1
        PUSH    HL  ;SAVE HL
        LD      H,B
        LD      L,C
        DEC     A  ;A CONTAINS REPEAT FACTOR-1
L1      ADD     HL,BC
        DEC     A
        CP      0
        JP      NZ,L1
        EX      DE,HL
        LD      BC,000AH
        ADD     HL,BC  ;NOW LOAD RESULT INTO VALUE FIELD
        LD      (HL),E
        INC     HL
        LD      (HL),D
        POP     HL  ;RETREIVE HL ADDR
        INC     HL
        INC     HL
        INC     HL
        PUSH    HL   ;STORE POINTER TO DEST#1 ADDR IN OP PACKET
        EX      DE,HL
        POP     HL
        LD      BC,0007
        ADD     HL,BC  ;HL NOW POINTS TO DEST#1 ADDR IN OP PACK
        EX      DE,HL  ;DE NOW POINTS TO DEST#1 ADDR IN RES PACK
        CALL    FORMPACK
        RET
REPLIC
        ;SINCE REPLICATE PERFORMS IN SAME MANNER AS CONSTANT
        ;SIMPLY CALL CONSTANT
        CALL    CONSTANT
        RET
INPUT
        RET
OUTPUT
        CALL    GET2OPER
        LD      HL,(OPERAND1)
        LD      E,L   ;SAVE OP1 IN E
        LD      HL,(OPERAND2)
        LD      D,L   ;SAVE OP2 IN D
        LD      HL,OUTBUF  ;LOAD HL WITH LOC OF OUTPUT BUFFER
REP1    LD      A,(HL)
        CP      20H    ;FIND 1ST BLANK AREA IN OUT BUFFER
        JP      Z,CONT3
        INC     HL
        JP      REP1
CONT3   LD      (HL),E  ;LOAD OP1 INTO OUTBUF
        INC     HL
        LD      (HL),D  ;LOAD OP2 INTO OUTBUF+1
        INC     HL
        LD      A,0DH  ;LOAD CARRIAGE RETURN
        LD      (HL),A
        INC     HL
        LD      A,0AH ;LINE FEED
        LD      (HL),A
        RET
SWITCH
        PUSH    HL
        PUSH    HL
        CALL    GET2OPER
        LD      HL,(OPERAND1)
        LD      E,L
        LD      HL,(OPERAND2)
        LD      D,L
        POP     HL
        LD      BC,000AH
        ADD     HL,BC
        LD      (HL),E
        INC     HL
        LD      A,0
        LD      (HL),A
        LD      A,D
        CP      "T"
        JP      NZ,ADDR2
ADDR1   EX      DE,HL
        POP     HL
        LD      BC,0005
        ADD     HL,BC
        INC     DE
        LD      A,(HL)
        LD      (DE),A
        INC     DE
        INC     HL
        LD      A,(HL)
        LD      (DE),A
        INC     DE
        LD      A,0
        LD      (DE),A
        INC     DE
        LD      (DE),A
        RET
ADDR2   EX      DE,HL
        POP     HL
        LD      BC,0007
        ADD     HL,BC
        LD      A,0
        INC     DE
        LD      (DE),A
        INC     DE
        LD      (DE),A
        INC     DE
        LD      A,(HL)
        LD      (DE),A
        INC     DE
        INC     HL
        LD      A,(HL)
        LD      (DE),A
        RET
ADDIT
        PUSH    HL
        PUSH    HL
        CALL    GET2OPER  ;GET THE TWO OPERANDS
        LD      HL,(OPERAND1)
        EX      DE,HL
        LD      HL,(OPERAND2)
        ADD     HL,DE  ;OP1 IN DE, OP2 IN HL
        EX      DE,HL  ;RESULT IN DE
        POP     HL  ;NOW GET ORIGINAL HL
        LD      BC,000AH
        ADD     HL,BC  ;HL NOW POINTS TO VALUE FIELD IN RES PACK
        LD      (HL),E
        INC     HL
        LD      (HL),D  ;RESULT NOW LOADED
        EX      DE,HL  ;SAVE HL IN DE
        POP     HL
        LD      BC,0005
        ADD     HL,BC  ;HL NOW POINTS TO DEST#1 ADDR IN OP PACK
        INC     DE     ;DE NOW POINTS TO DEST#1 ADDR IN RES PACK
        CALL    FORMPACK
        RET
SUBT
        RET
MULT
        PUSH    HL
        PUSH    HL
        CALL    GET2OPER
        LD      HL,(OPERAND1)
        LD      A,L   ;OPERAND1 IS REPEAT FACTOR
        DEC     A     ;A CONTAINS REPEATFACTOR - 1
        EX      DE,HL
        LD      HL,(OPERAND2)
        LD      B,H  ;PLACE DUPLICATE OF OPERAND2 IN
        LD      C,L  ;BE FOR REPEATED ADDITION
L2      ADD     HL,BC
        DEC     A
        CP      0
        JP      NZ,L2
        EX      DE,HL  ;STORE RESULTS IN DE
        POP     HL
        LD      BC,000AH  ;OFFSET TO GET TO VALUE LOC IN RES PACK
        ADD     HL,BC
        LD      (HL),E
        INC     HL
        LD      (HL),D  ;RESULT IS NOW LOADED INTO VALUE FIELD
        EX      DE,HL
        POP     HL
        LD      BC,0005
        ADD     HL,BC
        INC     DE
        CALL    FORMPACK
        RET
DIVIDE
        PUSH    HL
        PUSH    HL
        CALL    GET2OPER
        LD      HL,(OPERAND1)
        LD      B,L  ;OPERAND1 IN B
        LD      C,0  ;C WILL CONTAIN THE QUOTIENT
        LD      HL,(OPERAND2)
        LD      H,L    ;OPERAND2 IN H
        LD      L,B    ;OPERAND1 NOW IN L
L3
        LD      A,L
        CPL
        ADD     A,1    ;TWO'S COMPLEMENT REPRESENTATION OF OPERAND1
        ADD     A,H    ;ADDITION OF 2'COMP = SUBTRACTION
        LD      H,A
        INC     C    ;ADD 1 TO QUOTIENT
        CP      L
        JP      P,L3  ;REPEAT SUBTRACTION UNTIL REMAINDER LESS THAN
        JP      Z,L3  ;OPERAND1
        LD      E,C
        POP     HL
        LD      BC,000AH
        ADD     HL,BC
        LD      (HL),E
        INC     HL
        LD      A,0
        LD      (HL),A
        EX      DE,HL
        POP     HL
        LD      BC,0005
        ADD     HL,BC
        INC     DE
        CALL    FORMPACK
        RET
GREAT
        PUSH    HL
        PUSH    HL
        CALL    GET2OPER
        LD      HL,(OPERAND2)
        LD      A,L  ;LOAD OPERAND2 INTO ACC FOR COMPARISON
        LD      HL,(OPERAND1)
        CP      L    ;COMPARE OP2 TO OP1 (ACC-L) IS THE COMPARISON
        JP      Z,NOGREAT  ;FALSE COMPARISON OP2 = OP1
        JP      M,NOGREAT  ;FALSE COMPARISON OP2 < OP1
        JP      P,YESGREAT ;TRUE COMPARISON OP2 > OP1
NOGREAT
        LD      A,46H  ;LOAD F=FALSE IN ACC
        JP      CONT
YESGREAT
        LD      A,54H  ;LOAD T=TRUE
CONT    POP     HL
```

```
        LD      BC,000AH
        ADD     HL,BC
        LD      (HL),A
        INC     HL
        LD      A,0
        LD      (HL),A
        EX      DE,HL
        POP     HL
        LD      BC,0005
        ADD     HL,BC ;HL NOW POINTS TO DEST#1 ADDR IN OP PACK
        INC     DE    ;DE NOW POINTS TO DEST#1 ADDR IN RES PACK
        CALL    FORMPACK
        RET
LESS
        PUSH    HL
        PUSH    HL
        CALL    GET2OPER
        LD      HL,(OPERAND2)
        LD      A,L
        LD      HL,(OPERAND1)
        CP      L  ;COMPARE OP2 TO OP1 (ACC-L)
        JP      Z,NOLESS  ;OP2 = OP1
        JP      P,NOLESS  ;OP2 > OP1
        JP      M,YESLESS ;OP2 < OP1
NOLESS
        LD      A,46H
        JP      CONT1
YESLESS
        LD      A,54H
CONT1   POP     HL
        LD      BC,000AH
        ADD     HL,BC
        LD      (HL),A
        INC     HL
        LD      A,0
        LD      (HL),A
        EX      DE,HL
        POP     HL
        LD      BC,0005
        ADD     HL,BC
        INC     DE
        CALL    FORMPACK
        RET
NOTEQ
        ;NOT EQUAL TEST HAS BEEN PUT IN THE EQUAL PROCEDURE
        CALL    EQUAL
        RET
EQUAL
        PUSH    HL
        PUSH    HL
        LD      C,A  ;SAVE TYPE OF COMPARISON FOR LATER TEST
        CALL    GET2OPER
        LD      HL,(OPERAND2)
        LD      B,L   ;SAVE OPERAND2 IN B
        LD      HL,(OPERAND1)
        LD      A,C    ;RESTORE A WITH TYPE OF COMPARISON REQUIRED
        CP      "="    ;TEST IF = COMPARISON
        JP      Z,EQ
NEQ     LD      A,B  ;A CONTAINED # SO WE WANT NOTEQ TEST
        CP      L
        JP      Z,FALSELOAD  ;OP2 = OP1, THEREFORE FALSE
        JP      TRUELOAD
EQ      LD      A,B
        CP      L
        JP      NZ,FALSELOAD  ;OP2 <> OP1, THERFORE FALSE
TRUELOAD
        LD      A,54H
        JP      CONT2
FALSELOAD
        LD      A,46H
CONT2   POP     HL
        LD      BC,000AH
        ADD     HL,BC
        LD      (HL),A
        INC     HL
        LD      A,0
        LD      (HL),A
        EX      DE,HL
        POP     HL
        LD      BC,0005
        ADD     HL,BC
        INC     DE
        CALL    FORMPACK
        RET

;*****************
;  FORM RESULT PACKET
;*****************
FORMPACK
        LD      A,(HL)  ;LOAD LOW ORDER BYTE DEST#1 ADDR
        LD      (DE),A
        INC     HL
        INC     DE
        LD      A,(HL)  ;LOAD HIGH BYTE DEST#1 ADDR
        LD      (DE),A
        INC     HL
        INC     DE
        LD      A,(HL)  ;LOAD LOW BYTE DEST#2 ADDR
        LD      (DE),A
        INC     HL
        INC     DE
        LD      A,(HL)  ;LOAD HIGH BYTE DEST#2 ADDR
        LD      (DE),A
        RET
GET2OPER
        INC     HL
        LD      E,(HL)
        INC     HL
```

```
PROGRAM CONVERT;      (*   THIS PROGRAM IS USED TO ACCEPT THE DATA FLOW
                       NOTATION FROM THE KEYBOARD.  IT THEN CONVERTS
                       THE TEXT INPUT (FROM THE KEYBOARD) INTO THE
                       APPROPRIATE INTEGER AND TEXT FIELDS AND WRITES
                       INPUT NOTATION INTO DISK RECORDS DEFINED AS
                       THE "F" RECORD.  PROGRAM "DATAFLOW" THEN READS IN
                       THE RECORDS AND PROCESSES THE DATA FLOW NOTATION
                       WRITTEN BY THIS PROGRAM.  NOTE:  THIS PROGRAM
                       MUST!!!  BE USED TO CREATE THE DATA FLOW NOTATION
                       USED BY THE  "DATAFLOW" PROGRAM.       *)

 TYPE

        F =  RECORD
         NODENUM : INTEGER;
         OPERATOR : CHAR;
         NUMINPUTS,NUMCONSTANTS : INTEGER;
         CONSVALUE,NACKSIG : INTEGER;
         OUTPTS : ARRAY [1..2] OF
                RECORD
                OUTNODENUM : INTEGER;
                INPTNODENUM : INTEGER;
                END;
         ACKNODE1,ACKNODE2 : INTEGER;
         END;
 VAR

        S : STRING;
        C : CHAR;
        I,J,K,NUMNODES : INTEGER;
        LIST : INTERACTIVE;
        N1,N2,N3,N4,N5,N6,N7,N8,N9,N10 : INTEGER;
        FF : FILE OF F;

BEGIN
 WRITE('ENTER FILENAME TO BE WRITTEN TO ');
 READ(S);
 REWRITE(FF,S);
 RESET(LIST,'CONSOLE:');
 WRITELN(LIST,'ENTER # OF NODES TO INPUT ');
 READ(LIST,NUMNODES);
 CLOSE(LIST);
 FOR I := 1 TO NUMNODES DO
   BEGIN
        FF^ . NODENUM := I-1;

        RESET(LIST,'CONSOLE:');
        WRITELN(LIST,'ENTER NODE # ',I-1,' OPERATOR VALUE ');
        LD      D,(HL)
        EX      DE,HL
        LD      (OPERAND1),HL
        EX      DE,HL
        INC     HL
        LD      E,(HL)
        INC     HL
        LD      D,(HL)
        EX      DE,HL
        LD      (OPERAND2),HL
        RET
LOCK
        PUSH    HL
LK1     LD      HL,LOKLOC
        LD      A,(HL)
        CP      1
        JP      Z,LK1
        LD      A,1
        LD      (HL),A
        POP     HL
        RET
UNLOCK
        LD      HL,LOKLOC
        LD      A,0
        LD      (HL),A
        RET
        .END
        READLN(LIST,C);
        FF^ . OPERATOR := C;
        CLOSE(LIST);
        RESET(LIST,'CONSOLE:');

     (*  THE TEN  INPUTS ARE: 1 - #INPUTS
                              2 - #CONSTANTS
                              3 - CONSTANT VALUE
                              4 - NUMBER ACKNOWLEDGE SIGNALS REQ.
                              5 - OUTPUT NODE NUMBER
                              6 - OUTPUT NODE INPUT NUMBER
                              7 - SAME AS 5
                              8 - SAME AS 6
                              9 - NODE # TO SEND ACK. SIG. TO
                             10 - SAME AS 9                          *)


        WRITELN('ENTER THE TEN INPUTS: ');
        READ(LIST,N1,N2,N3,N4,N5,N6,N7,N8,N9,N10);
        FF^ . NUMINPUTS := N1;
        FF^ . NUMCONSTANTS := N2;
        FF^ . CONSVALUE := N3;
        FF^ . NACKSIG := N4;
        FF^ . OUTPTS[1].OUTNODENUM := N5;
        FF^ . OUTPTS[1].INPTNODENUM := N6;
        FF^ . OUTPTS[2].OUTNODENUM := N7;
        FF^ . OUTPTS[2].INPTNODENUM := N8;
        FF^ . ACKNODE1 := N9;
        FF^ . ACKNODE2 := N10;

        CLOSE(LIST);
        PUT(FF);
     END;
     CLOSE(FF,LOCK);

  END.  (* CONVERT *)
```

Appendix B

Multiprocessor User's Guide

This appendix contains a user's guide of all the software which comprises the data flow multiprocessor. Included in this appendix is an explanation of the function of each procedure or module, a procedure call list, how to use each module, and some important hardware idiosyncrasies.

I. Function of Each Module

HOST Software

The PASCAL Host program is called DATAFLOW. This program reads in the data flow notation (as in Figure 20), and converts the notation into activity templates (one template for each data flow instruction). Once the activity templates have been created, the program stores them in the activity store (shared memory locations E000 - E7FFH). Next, the program loads the slave processor software and enables the slave. Once the slave is enabled, the master processor is loaded and enabled. The slave software executes independently of both the host and master software. When it completes execution, the slave processor jumps to a ROM loader program and loops until it is re-enabled. When the master sottware terminates, it hands control back over to the host software. The final task of the host is then performed by dumping the output buffer. The contents of the output buffer will be displayed on the CRT in Hex format. The output will consist of two numbers (Hex) per line. There are several modules used by DATAFLOW to carry out the above functions. The following is a discussion of the function of each module contained in or called by DATAFLOW.

LISTFLOW. This is a PASCAL routine contained in the DATAFLOW program. Its function is to display the data flow notation read in by the DATAFLOW program. It displays this notation in the same format as the notation in Figure 20.

BLANKIT. This is an external assembly language procedure which must be linked together with DATAFLOW. This procedure resides in the program library called "N.LIBRARY". The function of this procedure is to initialize parts of shared memory (D800-F000H) with blanks so that previous program data will not have an affect on execution of the multiprocessor software.

ACTIVITYSTORE. This is a PASCAL procedure contained in DATAFLOW which calls an external assembly language procedure. The function of the procedure is to place the data flow notation into variables which are passed to the assembly language routine called FILLMEM.

FILLMEM. This is the external assembly procedure called by ACTIVITYSTORE. FILLMEM is linked with DATAFLOW by linking the library "N.LIBRARY". The function of this procedure is to convert the values passed in variables OP,N1,N2,N3...,N11, and J into activity templates. Each activity template represents one data flow instruction. The templates are stored in shared memory starting at locations E000H-E7FFH. Each template requires 16 contiguous bytes of storage so that each instruction lies on an even byte boundary.

LGOSLAVE. This is a PASCAL routine which reads into array PROG the slave software. Once the software is read, the procedure calls the external assembly procedure LOADSLAVE to load the software and enable the slave processor.

LOADSLAVE. This external assembly procedure is stored in the program library "N.LIBRARY". The function of this procedure is to load the slave software (stored in PROG) into high RAM (F800-FFFFH) of the slave processor. Once the software is loaded into the slave RAM, then the slave processor executes the software.

INIT. This is a PASCAL procedure which functions in the same manner as LGOSLAVE. INIT's function is to read the master processor software into PROG and then call the assembly language procedure LGORAM.

LGORAM. This is an external assembly procedure which is kept in "N.LIBRARY". The function of LGORAM is to load the master software (stored in PROG) in high RAM (F800 - FFFFH) of the master processor. Once the software is stored in RAM, the master processor executes. When the master software completes execution, it passes control back to the HOST.

DUMPOUT. This is an external assembly procedure also stored in "N.LIBRARY". The function of this procedure is to display the contents of the output buffer (D800-DFFFH) on the CRT. This procedure reads two values (HEX) and displays them on one line (CR and LF follow each pair of HEX values). At the completion of this procedure, the function of the

HOST is completed and control is handed back to UCSD PASCAL.

Program CONVERT

This is a stand-alone program which prompts the user for values which make up a data flow instruction. Once the user answers with proper values, the program stores the value in a record format. These records contain the data flow instructions (nodes) read and processed by the HOST program (DATAFLOW). This program must be used to create the data flow programs!

MASTER Software

The master software performs the "fetch" and "update" functions of the circular pipeline architecture of Chapter III. The specific functions of the master software are discussed in this section. The Master searches through the activity store and checks for enabled nodes. A node is enabled when all of its operands are present and when it has received the correct number of acknowledge signals. When an enabled node is found, Master calls the procedure ENABLE (all procedures are contained in Master). The ENABLE procedure takes the enabled node, converts it into an operation packet, and cleans the activity template. This means that the non-constant operands are removed and the acknowledge signal received value is reset to zero (thus making the node ready to be re-enabled). Master searches through all of the activity store until no more enabled nodes are found. At that time it checks the enabled node count value. If the value is zero, a "#" is written to END_FLAG (signaling the slave to stop), the master stops execution, and returns to the HOST software. If the value is non-zero, the procedure RESULT is called. The RESULT procedure reads the result packets (formed by the slave software) and passes the results to all required instructions in the activity store. The acknowledge signals are also updated. The Master continues executing in this manner until no more enabled nodes are present.

SLAVE Software

The Slave software reads the operation packets formed by the Master and performs the required operation on the operands. Once the operation has been performed on the operands, the Slave forms a result packet composed of the result of the operation and the destination addresses (where the result is sent). The Slave continues execution in this manner until a "#" appears in location F000H. When this happens, execution ceases and the slave processor jumps to ROM location 0 (location of the Slave ROM loader program - Figure 22).

II. Procedure Call Lists

The following lists define the main modules of the data flow multirocessor software and the procedures called by each module. The letters in parentheses after the module names denote whether the module was a PASCAL routine (P) or an assembly language routine (A).

HOST Software

Procedure	Calls
DATAFLOW(P)	LISTFLOW(P),BLANKIT(A), FILLMEM(A),LGOSLAVE(P), ACTIVITYSTORE(P),INIT(P), LOADSLAVE(A),LGORAM(P), DUMPOUT(A)
ACTIVITYSTORE(P)	FILLMEM(A)
LGOSLAVE(P)	LOADSLAVE(A)
INIT(P)	LGORAM(A)

MASTER Software
(all modules are assembly language)

Procedure	Calls
MASTER	ENABLE,RESULT
ENABLE	LOCK,OPPACK,UNLOCK
OPPACK	IENCT,CLEAN
RESULT	LOCK,UNLOCK,BLANK,ACKSIG DENBCT
ACKSIG	MULT

SLAVE Software
(all modules are assembly language)

Procedure	Calls
SLAVE	LOCK,OPERATION,UNLOCK
OPERATION	GET2OPER,FORMPACK

III. How to Use the Software

This section describes how to use the software. The MASTER and SLAVE software require no user response and their execution is entirely transparent to the user. Therefore, only the programs which require user interface will be discussed. All underlined text represents the system response.

```
1. X
2. EXECUTE WHAT FILE? #5:DGO
3. ENTER FILENAME OF DATA FLOW NOTATION #5:TEST5.DFN
4. WRITE INPUT LISTING [Y/N] N
5. DO YOU WANT THE SLAVE TO BE LOADED? [Y/N] Y
6. READING SLAVE SOFTWARE...
7. SLAVE SOFTWARE READ..NOW EXECUTE IT
8. SLAVE SOFTWARE EXECUTING
9. READING MASTER SOFTWARE
10. MASTER SOFTWARE READ .. NOW LOAD IT
11. MASTER SOFTWARE EXECUTING!!!!
12. DO YOU WANT THE OUTPUT BUFFER DUMPED? [Y/N] Y
13. 16 E
```

The "X" response in line 1 tells the UCSD PASCAL host that a file must be executed. The "#5:DGO" response in line 2 tells the system that DGO is the file to be executed. The response to the prompt of line 3 tells the DATAFLOW program (DGO) which data flow program to process. If the response of line 4 would have been "Y", the data flow program would have been displayed on the CRT in the same format as Figure 20. The response to line 5 must be "Y" in order for the program to be processed. However, a "N" should be used if a mistake were made in answering line 3. A "N" response to line 5 will cause fast termination of the DGO program. Lines 6 through 11 tell what the multiprocessor is doing since the SLAVE and MASTER software functions are transparent. Line 11 is somewhat misleading. It should only appear when the MASTER software has completed. The response to line 12 determines if the output should be dumped. A "N" value returns control back to UCSD PASCAL and a "Y" value dumps the buffer (values are in Hex). Once the buffer has been dumped, DGO returns control to UCSD PASCAL.

IV. Hardware Idiosyncrasies

Memory Board

The multiprocessor system uses the 64k memory board normally configured on the Intel Series II. When using this board, both banks of memory must be switched on. The banks are switched on by means of two dip switches. Both switches must be on!!

Bus Priorities

The disk interface must occupy the first location on the bus. This gives it the highest priority on the bus (Figure 28). The next boards should be the slave processor, followed by the master processor, followed by the RAM memory board. The slave must have higher priority than the master since its operation requires less time than the master. The master is busier than the slave and locks out shared memory for longer periods. If the master had a higher priority than the slave, it would have the potential of locking out the slave processor.

Processor RAM

The on-board RAM address space on each processor must be assigned to locations F800-FFFFH. This is done by jumpering pins 117 to 121. If the RAM address space is assigned to any of the other three locations, it will prevent UCSD PASCAL from executing.

Power Supply Adjustments

The power supply must be adjusted for a multiprocessor configuration (5 boards are present on the multi-bus). When configured for the system described in this thesis, the number of boards and their power requirements put a heavy load on the power supply. The multi-bus must receive at least +4.75V in order for the signal to probagate through all boards. The adjustment is made by adjusting the blue "R15" switch on the elevated circuit board of the power supply.

Vita

Captain Michael W. Bray was born on June 19, 1954 in Fort Hood, Texas. In 1972, he graduated from O. Perry Walker Senior High School in New Orleans, Louisiana. He attended Louisiana State University and received a Bachelor of Science Degree in Computer Science in 1976. Following graduation, he was assigned to the Air Defense Weapons Center (ADWC) at Tyndall AFB, Florida. In 1978, he was assigned to the 475th Test Squadron, also at Tyndall AFB. At the 475th Test Squadron, he served as the Deputy Chief of the Computer Support Branch. He entered the Air Force Institute of Technology in June 1980.

Permanent Address:
5238 Woodside Dr.
Baton Rouge, La.
70808

REPORT DOCUMENTATION PAGE

READ INSTRUCTIONS BEFORE COMPLETING FORM

1. REPORT NUMBER AFIT/GCS/EE/81D-5	2. GOVT ACCESSION NO. AD-A115616	3. RECIPIENT'S CATALOG NUMBER
4. TITLE (and Subtitle) THE DESIGN AND IMPLEMENTATION OF A DATA FLOW MULTIPROCESSOR		5. TYPE OF REPORT & PERIOD COVERED MS Thesis 6. PERFORMING ORG. REPORT NUMBER
7. AUTHOR(s) Michael W. Bray, Captain USAF		8. CONTRACT OR GRANT NUMBER(s)
9. PERFORMING ORGANIZATION NAME AND ADDRESS Air Force Institute of Technology (AFIT-EN) Wright-Patterson AFB, Ohio 45433		10. PROGRAM ELEMENT, PROJECT, TASK AREA & WORK UNIT NUMBERS
11. CONTROLLING OFFICE NAME AND ADDRESS Air Force Institute of Technology (AFIT-EN) Wright-Patterson AFB, Ohio 45433		12. REPORT DATE December 1981 13. NUMBER OF PAGES 156
14. MONITORING AGENCY NAME & ADDRESS(if different from Controlling Office)		15. SECURITY CLASS. (of this report) **UNCLASSIFIED** 15a. DECLASSIFICATION/DOWNGRADING SCHEDULE

16. DISTRIBUTION STATEMENT (of this Report)

Approved for public release; distribution unlimited

17. DISTRIBUTION STATEMENT (of the abstract entered in Block 20, if different from Report)

[illegible]
LYNN E. WOLAVER
Dean for Research and Professional Development

4 JUN 1982

18. SUPPLEMENTARY NOTES

Approved for public release; IAW AFR 190-17

~~Frederic C. Lynch, Major USAF~~
~~Director of Public Affairs~~

Dean for Research and Professional Development
Air Force Institute of Technology (ATC)
Wright-Patterson AFB, OH 45433

19. KEY WORDS (Continue on reverse side if necessary and identify by block number)

Data Flow
Small Board Computers
Multiprocessing
Data Driven Language

20. ABSTRACT (Continue on reverse side if necessary and identify by block number)

This report presents a data flow multiprocessor designed and implemented with standard laboratory microcomputer boards. Intel SBC 80/20 small board computers were used since their design supported a multibus structure required for multiprocessing. Current data flow techniques were researched in order to find a technique that could be implemented through software. A packet communication architecture was chosen for implementation since other data flow techniques required specialized hardware.

The requirements of the multiprocessor were defined using structured analysis techniques. These requirements were then translated into structured modules. The software modules were then implemented and tested in a top down approach. The data flow multiprocessor software was tested by executing complete data flow programs. The results of the tests demonstrated the functionality of the multiprocessor. However, the multiprocessor software was limited in some respects. The memory allocated for the storage of data flow programs limited the maximum number of data flow instructions that could be represented to only 128. The mathematical operations were also limited in that only 8 bit computations were allowed.

B

FILMED

7-8